Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська кваліфікаційна робота

на тему: «Розробка TRUE FIRST PERSON SHOOTER гри "Rock In A Hard Place: Lonesome Road" з використанням платформи Unreal Engine 5»

Виконав: студент IV курсу
групи 1ПІ-21б
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Пахолюк В. Б.
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ Катєльніков Д. І.
(прізвище та ініціали)
Рецензент: к.т.н., доц. каф. ОТ Тарновський М. Ю.
(прізвище та ініціали)

Допущено до захисту
Зав. кафедри ПЗ Романюк О.Н.
«09» 06 2025 р.

ВНТУ– 2025

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший бакалаврський
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О. Н.
"21" березня 2025 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Пахолюку Владиславу Борисовичу

1. Тема роботи – «розробка TRUE FIRST PERSON SHOOTER гри "Rock In A Hard Place: Lonesome Road" з використанням платформи Unreal Engine 5».

Керівник роботи: Катєльніков Денис Івнович, к.т.н., доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від „20" березня 2025 р. № 97

2. Строк подання студентом роботи 2 червня 2025 р.

3. Вихідні дані до роботи: середовище розробки – Unreal Engine 5, Visual Studio; мова програмування – C++.

4. Зміст розрахунково–пояснювальної записки: вступ; обґрунтування вибору методу розробки та постановка задач дослідження; розробка структури, методу, моделі та алгоритмів роботи програмного продукту; розробка програмного забезпечення; тестування програми; список використаних джерел; додатки.

5. Перелік графічного матеріалу: Титульний слайд, Актуальність теми, Мета, об'єкт та предмет дослідження, Задачі дослідження, Наукова новизна, Практична цінність одержаних результатів, Порівняльний аналіз аналогів, Блок-схема алгоритму створення тестів, Блок-схема алгоритму генерування текстових та графічних компонентів, Тестування програми (створення власних тестів), Тестування програми (проходження тестування), Тестування програми (скидання даних користувача), Тестування програми (звуковий супровід при вивченні нових слів), Апробація та публікації результатів роботи.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Катєльніков Д. І. к.т.н., доцент кафедри ПЗ	24.03.25	02.06.25

7. Дата видачі завдання 24 березня 2025 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз завдання і вибір методу вирішення поставленої задачі дослідження	26.03.2025 - 29.03.2025	вик
2	Вибір архітектури програмного компоненту	01.04.2025 - 09.04.2025	вик
3	Розробка структури графічного інтерфейсу „Pollaung”	10.04.2025 - 19.04.2025	вик
4	Вибір середовища та мови розробки	20.04.2025 - 24.04.2025	вик
5	Розробка модулів програми	27.04.2025 - 11.05.2025	вик
6	Тестування програми	11.05.2025 - 15.05.2025	вик
7	Оформлення матеріалів до захисту БКР	16.05.2025 - 30.05.2025	вик

Студент _____ (підпис) **Пахолюк В.Б.** (прізвище та ініціали)

Керівник бакалаврської кваліфікаційної роботи _____ (підпис) **Катєльніков Д. І.** (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з 53 сторінок формату А4, на яких представлено 12 рисунків, 2 таблиці, а список використаних джерел містить 20 найменувань.

Бакалаврська кваліфікаційна робота присвячена розробці гри у жанрі TRUE FIRST PERSON SHOOTER під назвою "Rock In A Hard Place: Lonesome Road". Тема роботи є актуальною у зв’язку зі зростанням попиту на високоякісні ігрові продукти, що поєднують інноваційні механіки, захопливий сюжет та реалістичну графіку.

Встановлено об’єкт, предмет, завдання та методи дослідження. Система призначена для створення захопливого ігрового досвіду з особливим акцентом на реалістичну фізику балістики зброї, що забезпечує глибше занурення гравця у сюжет та ігровий процес. Для реалізації мети було розроблено гру у жанрі true first person shooter.

Розробку гри виконано на основі ігрового рушія Unreal Engine із використанням мови програмування C++ у середовищі розробки Visual Studio. Для реалізації графічних і фізичних ефектів використовувалися вбудовані засоби Unreal Engine. У результаті виконання бакалаврської кваліфікаційної роботи створено працездатний ігровий прототип, проведено тестування його функціоналу та підготовлено документацію для розробників і кінцевих користувачів.

Ключові слова: unreal engine, шутер, гра.

ABSTRACT

The bachelor's qualification work consists of 53 A4 pages, featuring 12 illustrations and 2 tables, with a bibliography of 20 references.

The thesis is dedicated to the development of a game in the TRUE FIRST PERSON SHOOTER genre titled Rock In A Hard Place: Lonesome Road. The topic is relevant due to the increasing demand for high-quality gaming products that combine innovative mechanics, an engaging storyline, and realistic graphics.

The object, subject, objectives, and research methods were identified. The system is designed to create an immersive gaming experience with a particular focus on realistic weapon ballistics physics, ensuring deeper player engagement in the story and gameplay. To achieve this goal, a game in the true first-person shooter genre was developed.

The game development was carried out using the Unreal Engine game engine and the C++ programming language within the Visual Studio development environment. Built-in Unreal Engine tools were employed to implement graphical and physical effects. As a result of the bachelor’s thesis, a functional game prototype was created, its functionality was tested, and documentation for developers and end-users was prepared.

Keywords: Unreal Engine, shooter, game.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. Сучасна ігрова індустрія демонструє стрімке зростання та стабільно займає провідні позиції серед прибуткових напрямів цифрових розробок. У цьому середовищі особливу нішу посідає жанр шутерів від першої особи. Проте окремо варто виокремити піджанр true first-person shooter (true FPS), що зосереджений на максимальному реалізмі та зануренні гравця в ігровий процес.

На відміну від класичних FPS, true FPS-ігри забезпечують гравцю повноцінне візуальне відображення тіла персонажа у перспективі першої особи — включно з ногами, торсом, рухами голови та інерцією камери. Такий підхід дозволяє досягти більшої правдоподібності, глибшої взаємодії з ігровим середовищем і значно сильнішого ефекту присутності.

Попит на true FPS постійно зростає, оскільки гравці все більше цінують не лише динамічний геймплей, а й естетику руху, увагу до деталей, фізику та реалістичну взаємодію з простором. Ці аспекти стали актуальними завдяки розвитку сучасних ігрових рушіїв, які дають змогу реалізовувати складну анімацію, просунуту кінематику та адаптивні камери.

Обрана тема дослідження є актуальною, оскільки створення true FPS вимагає комплексного підходу до розробки: від дизайну рівнів та програмування керування до фізичної симуляції та побудови анімацій. Реалізація такого проєкту дозволяє не лише заглибитись у специфіку ігрової розробки, а й опанувати сучасні інструменти, що широко використовуються у професійному середовищі.

Зв’язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою роботи є підвищення реалістичності 3D-симулятора сучасного бою за рахунок розробки програмної системи візуалізації з урахуванням балістики зброї та алгоритмів обчислення взаємодії кулі з ігровим середовищем.

Основними задачами дослідження є:

- проаналізувати існуючі відеоігри-аналоги які використовують технологію True First Person;
- виконати дослідження структури гри:
    - рух персонажа;
    - управління зброєю;
    - балістика куль.
- розробити алгоритми обрахунку поведінки куль;
- розробити програмний продукт;
- провести тестування розробленої відеогри.

Об’єкт дослідження: процес розробки True First Person шутеру «Rock In A Hard Place» за допомогою Unreal Engine 5.

Предмет дослідження: методи та засоби обчислення балістики зброї та кулі з ігровим середовищем.

Методи дослідження. У процесі досліджень використовувались: методи тривимірної комп’ютерної графіки для відтворення реалістичного 3D-симулятора сучасного бою; методи стереометрії для обчислення траєкторії руху кулі; теорія графів для розробки логіки роботи основної бойової механіки; комп’ютерне моделювання для аналізу та перевірки отриманих теоретичних положень.

Наукова новизна отриманих результатів.

Подальшого розвитку набув метод розробки True First Person-шутерів, який, на відміну від існуючих, не лише імітує постріл візуальним ефектом, а й виконує високоточні розрахунки траєкторій куль, моделює віддачу зброї, знос та інші аспекти, що забезпечують глибше занурення гравця у віртуальне середовище.

Практична цінність отриманих результатів.

Практична цінність одержаних результатів полягає в тому, що на основі отриманих в бакалаврській кваліфікаційної роботі теоретичних положень

запропоновано алгоритми та розроблено програмні засоби відеогри «Rock In A Hard Place».

Особистий внесок здобувача. Усі наукові результати, викладені у бакалаврській кваліфікаційній роботі і в науковій публікації[1], отримані особисто автором.

Апробація матеріалів бакалаврської кваліфікаційної **роботи.** Основні положення бакалаврської кваліфікаційної роботи доповідалися та обговорювалися на IV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів і студентів "Комп`ютерні ігри та мультимедіа як інноваційний підхід до комунікації -2024", Одеса,[1].

Публікації. Основні результати досліджень опубліковано у 1 науковій праці: 1 – у матеріалах конференцій.

1 ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДУ РОЗРОБКИ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Аналіз стану технологій симуляції озброєння у відеоіграх

Сучасний етап розвитку ігрових та симуляційних технологій вимагає постійного вдосконалення методів відтворення механік озброєння, зокрема у тривимірному просторі. Традиційні підходи до симуляції зброї, хоча й залишаються основою для розробки багатьох ігор і тренажерів, часто мають низький рівень реалістичності, обмежені фізичними моделями та недостатньою інтерактивністю. У цьому контексті, застосування сучасних ігрових рушіїв, таких як Unreal Engine, дозволяє значно підвищити рівень деталізації й точності симуляції озброєння.

Ключовим фактором розвитку цих технологій є збільшення обчислювальної потужності та вдосконалення графічних процесорів, що дозволяє створювати реалістичні фізичні моделі поведінки озброєння. Сучасні системи дозволяють виконувати високоточні розрахунки траєкторій куль, моделювати віддачу зброї, знос та інші аспекти, що забезпечують глибше занурення гравців у віртуальне середовище.

Реалістичність тривимірної симуляції озброєння є важливим аспектом у багатьох галузях, таких як ігрова індустрія, військові тренажери та освітні симуляції. Точне відтворення балістичних параметрів, поведінки зброї та її фізичних характеристик дозволяє створювати не лише захопливий ігровий досвід, але й тренажери для відпрацювання навичок поводження зі зброєю у професійній сфері.

Однією з основних переваг використання сучасних технологій є можливість інтеграції графічних інтерфейсів для налаштування зброї, аналізу траєкторій куль, а також візуалізації впливу середовища на фізику об'єктів. Це забезпечує кращу взаємодію користувачів із системою та розширює функціональні можливості симуляторів.

Також важливою є адаптація програмного забезпечення під різні платформи, що дозволяє зробити симуляції доступними для широкого кола користувачів. Використання таких рушіїв, як Unreal Engine, забезпечує високий рівень сумісності та дозволяє створювати багатоплатформні рішення, які можуть бути застосовані в різних контекстах.

Отже, аналіз стану технологій симуляції озброєння у тривимірному середовищі підтверджує актуальність розробки програмного забезпечення, яке забезпечує високу точність, реалістичність і взаємодію користувача з віртуальним озброєнням. Розробка інноваційних модулів, таких як моделювання балістики, управління зброєю та інтерактивні інтерфейси, є важливим кроком у вдосконаленні існуючих рішень та створенні нових підходів до симуляції.

1.2 Порівняльний аналіз аналогів

Шутери є одним із найпопулярніших жанрів у ігровій індустрії, що постійно вдосконалюється завдяки розробці нових механік, взаємодії із зброєю та створенням захопливого ігрового процесу. Розробка програмного забезпечення для шутерів забезпечує реалістичність бойових дій, збалансованість озброєння та можливість інтерактивної взаємодії гравця з ігровим світом.

До найвідоміших шутерів, які визначили розвиток жанру, належать:

- Call of Duty - культова серія шутерів, що охоплює різні історичні та сучасні конфлікти;
- Battlefield - гра, яка відома своїми масштабними багатокористувацькими битвами та акцентом на командну гру;
- Counter-Strike: Global Offensive (CS:GO) - класичний шутер, який залишається одним із найпопулярніших завдяки своїй змагальній структурі;
- DOOM - легендарний шутер, який став родоначальником жанру й задає темп динамічному ігровому процесу;

- Rainbow Six Siege — тактичний шутер, що поєднує бойові дії з елементами стратегічного планування та командної взаємодії.

Одним із прикладів реалізації реалістичної взаємодії гравця зі зброєю є серія ігор Call of Duty (рис. 1.1) – популярний шутер від першої особи, розроблений компаніями Infinity Ward, Treyarch та Sledgehammer Games. Гра забезпечує високу деталізацію зброї, реалістичну фізику стрільби та анімації перезарядки, що створює глибоке занурення в бойовий процес і підвищує рівень ігрового реалізму.

Рисунок 1.1 – Вид від першої особи в Call of Duty

Прикладом гри, що забезпечує реалістичну взаємодію гравця зі зброєю, є серія Battlefield (рис. 1.2) – тактичний шутер від першої особи, розроблений студією DICE та виданий компанією Electronic Arts. Завдяки продуманій фізиці озброєння, системі руйнування оточення та масштабним багатокористувацьким боям, Battlefield створює глибокий ігровий досвід, наближений до реальних бойових умов.

Рисунок 1.2 – Вид від першої особи в Battlefield

Ще одним прикладом гри, що забезпечує реалістичну взаємодію гравця зі зброєю, є Counter-Strike: Global Offensive (рис. 1.3) – популярний тактичний шутер від першої особи, розроблений компаніями Valve та Hidden Path Entertainment. Гра відзначається точною моделлю стрільби, унікальними характеристиками кожного виду зброї та змагальним ігровим процесом, що робить її однією з найпопулярніших кіберспортивних дисциплін.

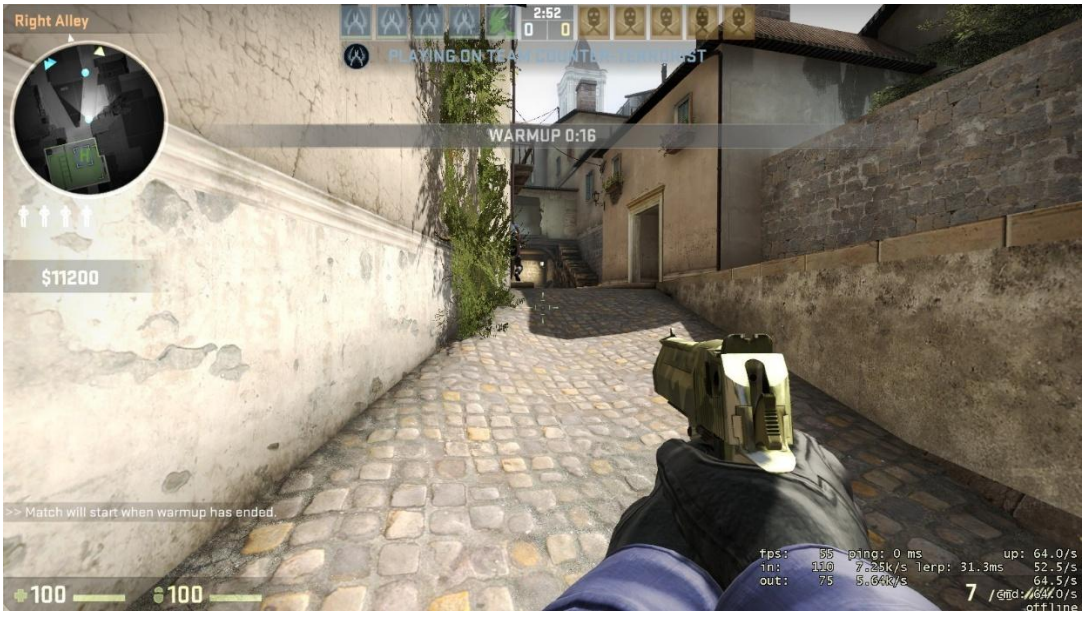

Рисунок 1.3 – Вид від першої особи в Counter-Strike: Global Offensive

Прикладом гри, що пропонує інтенсивний бойовий досвід та унікальну систему взаємодії зі зброєю, є DOOM (рис. 1.4) – легендарний шутер від першої особи, створений студією id Software. Гра вирізняється високою динамікою, аркадним стилем ведення бою та широким вибором озброєння, що сприяє агресивному ігровому процесу та безперервній динаміці сутичок.

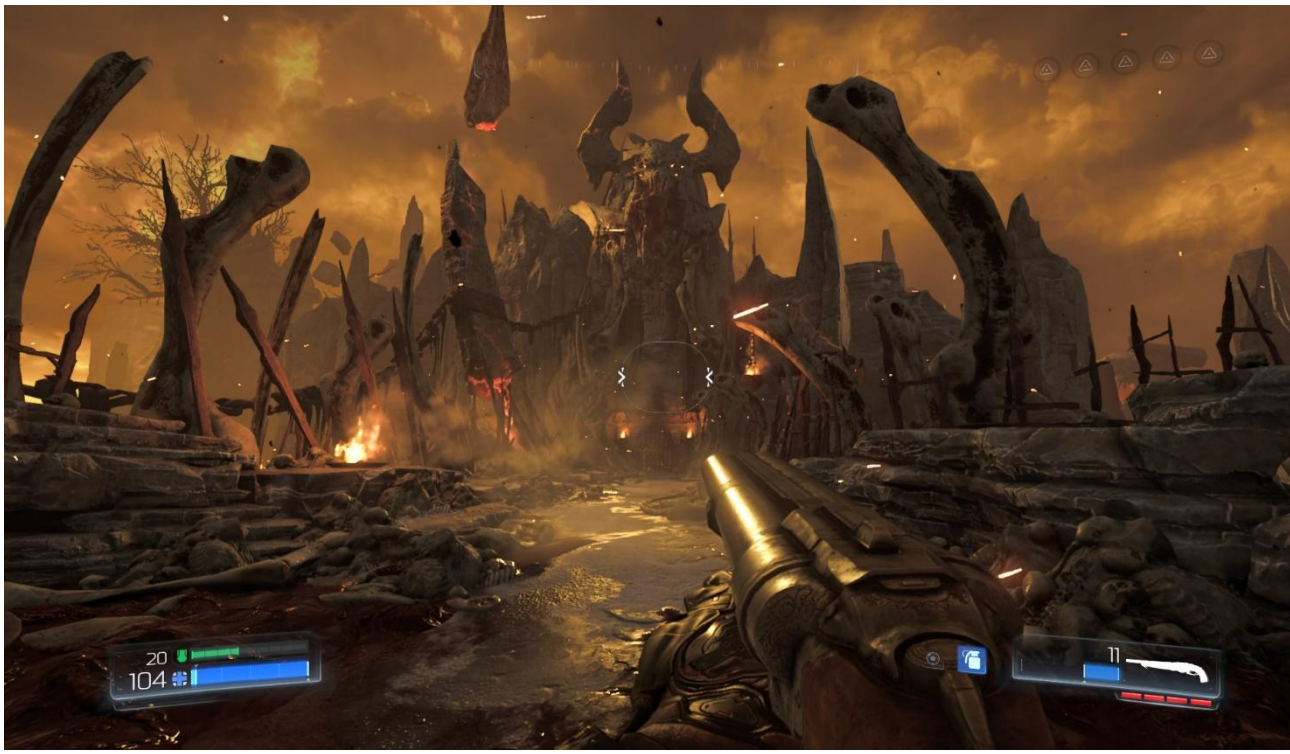

Рисунок 1.4 – Вид від першої особи в Doom

Rainbow Six Siege (рис. 1.5) — це тактичний шутер від першої особи, розроблений компанією Ubisoft. Гра зосереджена на командній взаємодії та тактичному підході до ведення бою, пропонуючи унікальних операторів із різними здібностями та спеціалізаціями. *Rainbow Six Siege* відзначається руйнованим оточенням, стратегічним використанням укриттів і спецзасобів, що створює глибокий ігровий процес та забезпечує високу реіграбельність.

Рисунок 1.5 – Вид від першої особи в Rainbow Six Siege

Розробка програмних засобів для симуляції озброєння в шутер-іграх, таких як Rainbow Six Siege, вимагає поєднання навичок у 3D-моделюванні, програмуванні та дизайні інтерфейсів. Основна частина розробки полягає у створенні системи, яка забезпечує реалістичну балістику, керування зброєю та зручний інтерфейс для гравця.

Симуляція балістики включає створення алгоритмів, що враховують такі фактори, як швидкість кулі, її напрямок, гравітацію та опір повітря. Це може бути реалізовано через стандартні компоненти рушія, наприклад, Projectile Movement Component в Unreal Engine, або за допомогою власних алгоритмів для детальнішого контролю.

Керування зброєю включає розробку механік для прицілювання, віддачі, темпу стрільби та перезарядки. Окрім цього, важливим є точне відображення кожного виду зброї у вигляді анімацій та інтерактивних елементів.

Результати порівняння наведено у таблиці 1.1.

Таблиця 1.1 – Порівняльна таблиця аналогів

Критерії	Call of Duty	Battlefield	Counter-Strike: Global Offensive	DOOM	Rainbow Six Siege	Rock In A Hard Place
Можливість змінювати оптику	+	-	-	-	-	+
Симуляція балістики	-	+	-	-	-	+
Вид від справжньої першої особи	-	-	-	-	-	+
Динамічне управління персонажем	-	-	-	-	+	+
Можливість прицілювання	-	-	-	-	+	+
Загальна оцінка	20%	20%	0%	0%	40%	100%

Відповідно до таблиці порівняльних характеристик, розробка власного шутера є доцільною, оскільки він зможе усунути недоліки існуючих ігор та запропонувати розширені можливості симуляції балістики, управління персонажем і взаємодії зі зброєю.

Отже, Реалізація таких механік сприятиме створенню більш реалістичного та захопливого ігрового процесу, що зробить гру конкурентоспроможною серед сучасних тактичних шутерів.

1.3 Аналіз методів розв’язання задачі

Розробка нового тактичного шутера вимагає комплексного підходу до вирішення завдань, пов’язаних із створенням реалістичної бойової механіки, симуляцією балістики та інтерактивним середовищем. Існують різні підходи до розробки подібних ігор, кожен з яких має свої переваги та недоліки. У цьому розділі проаналізовано найбільш ефективні методи створення таких ігрових систем.

Одним із ключових аспектів розробки є реалізація реалістичної механіки руху та управління персонажем. Для цього можна використовувати анімаційні системи, що базуються на процедурній генерації рухів або технології motion capture. Використання motion capture забезпечує високу точність відображення анімаційних рухів, але потребує значних ресурсів та спеціального обладнання. Процедурна генерація, навпаки, дозволяє динамічно адаптувати рухи персонажа до ігрового середовища, проте може поступатися в деталізації.

Інший важливий аспект – симуляція балістики. Традиційні системи використовують спрощену модель стрільби, що базується на влучанні в хитбокси без урахування фізичних факторів. Більш реалістичний підхід передбачає розрахунок траєкторії куль із врахуванням сили тяжіння, опору повітря та пробивної здатності. Наприклад, методи фізичного моделювання дозволяють створити більш правдоподібний досвід стрільби, проте їх реалізація вимагає значних обчислювальних ресурсів.

Також варто розглянути можливість зміни оптики та налаштувань зброї. Для цього можуть бути використані модульні системи кастомізації, що дозволяють гравцям налаштовувати приціли, глушники, рукоятки та інші елементи зброї. Це збільшує тактичну варіативність і дає змогу гравцям підлаштовувати зброю під власний стиль гри.

Після аналізу переваг і недоліків кожного підходу було вирішено використати комбінований метод: впровадження процедурної генерації анімацій у поєднанні з технологіями motion capture, реалізацію реалістичної балістики з урахуванням фізичних параметрів та розробку гнучкої системи кастомізації

зброї. Це забезпечить високу якість ігрового процесу та створить унікальну платформу з широкими можливостями для гравців.

1.4 Постановка задач

Проаналізувавши переваги та недоліки існуючих тактичних шутерів, було визначено наступні завдання, які необхідно виконати для успішної розробки гри:

- розробити алгоритм стрільби;
- розробити алгоритми обчислення балістики куль;
- розробити модуль управління персонажем;
- розробити модуль управління зброєю;
- реалізувати програмний продукт;
- провести тестування розробленої відеогри згідно поставлених задач.

1.5 Висновки

У першому розділі було розглянуто стан розвитку тактичних шутерів та їх ключові особливості. Були проаналізовані такі ігри, як Call of Duty, Battlefield, Counter-Strike: Global Offensive, DOOM та Rainbow Six Siege. Кожна з них має власні унікальні механіки, що визначають її місце на ринку та серед гравців.

Проаналізувавши переваги та недоліки цих ігор, було сформульовано ключові завдання для подальшої розробки власного тактичного шутера. До них належать розробка реалістичної балістики, впровадження динамічного керування персонажем, можливість змінювати оптику та вплив оточення на ігровий процес. Таким чином, було доведено доцільність створення власного ігрового рішення, яке буде поєднувати сильні сторони існуючих проектів та пропонувати унікальні геймплейні можливості. На основі отриманої інформації було сформовано перелік задач, необхідних для розробки гри.

2 РОЗРОБКА СТРУКТУРИ, МЕТОДУ, МОДЕЛІ ТА АЛГОРИТМІВ РОБОТИ ПРОГРАМНОГО ПРОДУКТУ

2.1 Аналіз вхідних даних системи

Для розробки шутер-гри "Rock In A Hard Place: Lonesome Road" буде використаний рушій Unreal Engine 5 у поєднанні з мовою програмування C++. Unreal Engine 5 – це сучасний і потужний ігровий рушій, який підтримує розробку ігор з високим рівнем деталізації завдяки технологіям Nanite, Lumen та іншим інноваціям, забезпечуючи реалістичну графіку й динамічне освітлення. C++ – це високопродуктивна об'єктно-орієнтована мова програмування, яка широко використовується для створення ігор завдяки прямому контролю над пам'яттю та високій швидкодії. Unreal Engine 5 надає можливість глибокої інтеграції C++ для створення складних ігрових механік, систем штучного інтелекту та оптимізації продуктивності. Поєднання можливостей Unreal Engine 5 і потужності C++ дозволить створити шутер-гру з високою якістю графіки, реалістичною фізикою та захопливою ігровою механікою, що відповідає сучасним вимогам індустрії.

Розробка шутера "Rock In A Hard Place: Lonesome Road" передбачає створення кількох ключових модулів. Першим буде модуль управління персонажем, який відповідатиме за обробку введення користувача та перетворення його в дії в ігровому світі. Цей модуль реалізовуватиме рухи персонажа, стрибки, ухилення та взаємодію з оточенням, забезпечуючи зручне та чітке керування за допомогою клавіатури, миші або геймпада. Далі буде розроблений модуль зброї, який відповідатиме за створення системи стрільби, перемикання між різними видами зброї, перезарядку, а також реалізацію фізичних та візуальних ефектів пострілу. Цей модуль забезпечить реалістичну поведінку зброї та глибоку взаємодію зі світом гри.

Розробка модулів управління персонажем та системи зброї потребує глибокого розуміння тривимірної взаємодії, фізики віртуального середовища та програмування на C++ в контексті Unreal Engine 5. Для реалізації цих модулів будуть використані різноманітні інструменти та можливості рушія, зокрема

системи Blueprints для швидкого прототипування, Animation Blueprint для налаштування рухів персонажа, а також модулі Chaos Physics для реалістичної обробки фізики стрільби та зіткнень. Розроблені модулі пройдуть етапи тестування на коректність роботи, продуктивність та інтеграцію зі складовими гри.

Отже, створення модулів управління та зброї для гри "Rock In A Hard Place: Lonesome Road" є комплексним завданням, яке вимагає детального проєктування, реалізації та перевірки. Маючи у своєму розпорядженні потужний рушій Unreal Engine 5, можливості мови C++ та сучасні інструменти розробки, можна створити стабільну, захоплюючу та технічно досконалу гру, яка відповідатиме очікуванням користувачів та сучасним стандартам індустрії інтерактивних розваг.

2.2 Розробка інтерфейсу програмного додатку

При розробці інтерфейсу гри було обрано шлях мінімалістичного дизайну, з акцентом на ефективність і зручність користування, щоб не перевантажувати гравця зайвими елементами. Ідея полягала в тому, щоб інтерфейс не відволікав від основного ігрового процесу, а підтримував його, забезпечуючи максимальну концентрацію на дії та навколишньому світі гри.

Основним елементом дизайну стала нейтральна кольорова палітра, яка включає відтінки сірого, білого та чорного. Ці кольори не тільки створюють стриману атмосферу, але й сприяють більш комфортному сприйняттю візуальних елементів під час напружених боїв. Сірі відтінки дозволяють візуально «поглинути» елементи інтерфейсу, роблячи їх менш агресивними, що дозволяє зберігати увагу гравця на основних подіях гри, а не на додаткових вікнах або індикаторах.

Чорний колір використовується для акцентування важливих елементів, таких як індикатори здоров’я, боєкомплекту чи цілі, а білий додає контрасту для важливих підказок і текстових повідомлень. Завдяки цьому балансі, інтерфейс виглядає органічно та не привертає зайву увагу, залишаючи гравця в світі гри.

Для збереження зручності та функціональності під час інтенсивних боїв було впроваджено динамічні елементи інтерфейсу, які мінімалістично спливають лише в потрібний момент, наприклад, під час перезарядки чи взаємодії зі зброєю. Це дозволяє зберігати інтерфейс чистим, не втрачаючи при цьому важливу інформацію для гравця. Також візуальні підказки, як-от підсвічування предметів або анімації, вказують на можливість взаємодії, коли це необхідно, не порушуючи загальну атмосферу гри.

Цей підхід дозволяє інтерфейсу бути абсолютно непомітним у моменти високої напруги, надаючи лише ту інформацію, яка потрібна гравцеві для прийняття оперативних рішень. Однак у спокійніших моментах гравець завжди має доступ до всіх необхідних даних для комфортної гри. Це створює динамічний і адаптивний інтерфейс, який підлаштовується під потреби гравця, зберігаючи баланс між інформативністю та зручністю.Приклад головного меню зображено на рисунку 2.1.

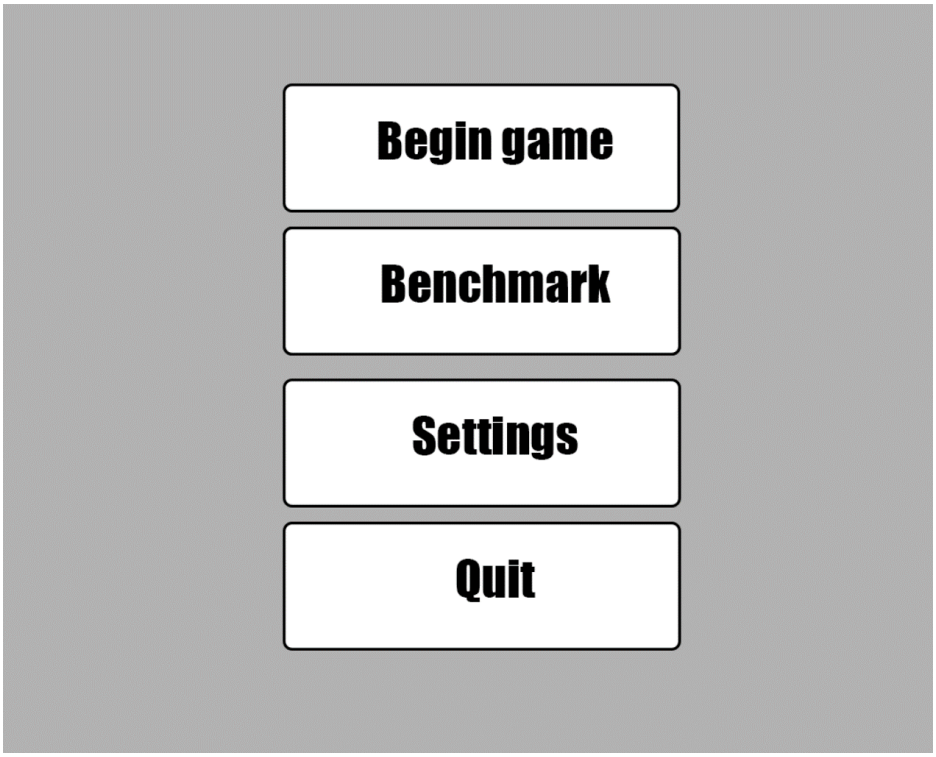

Рисунок 2.1 – Головне меню

У головному меню гри доступні кілька основних опцій для гравця:

1. Begin Game — розпочинає гру, ініціюючи запуск основного ігрового процесу.
2. Benchmark — запускає тестування гри, дозволяючи оцінити продуктивність та оптимізацію на різних налаштуваннях.
3. Settings — відкриває меню налаштувань, де гравець може налаштувати параметри графіки, звуку та інші функції гри для комфортної гри.
4. Quit — виходить з гри, завершуючи поточну сесію.

Ці функції забезпечують зручний доступ до основних можливостей гри та дозволяють користувачу швидко налаштувати досвід гри відповідно до своїх уподобань.

Оскільки гра передбачає активну взаємодію зі зброєю, для полегшення цього процесу були створені спеціальні візуальні елементи, які інформують гравця про можливість взаємодії. Ці елементи включають індикацію доступних предметів для взаємодії, такі як підсвічування чи анімації, що вказують на місця, де гравець може підняти зброю чи змінити її. Візуальні підказки дозволяють гравцеві швидко зорієнтуватися в інтерфейсі і не витрачати час на пошук важливих об'єктів у динамічному ігровому середовищі. На рисунку 2.2 зображено предмет до наведення на нього курсору, на рисунку 2.3 зображено предмет після наведення на нього курсору.

Рисунок 2.2 – Предмет до наведення на нього

Рисунок 2.3 – Предмет після наведення на нього

Розробка графічного інтерфейсу гри була проведена у середовищі Unreal Engine 5 із використанням технології Blueprints для створення інтерфейсу. Blueprints є візуальним скриптовим інструментом Unreal Engine, що дозволяє ефективно створювати інтерфейси без необхідності написання великої кількості коду. Це дозволяє швидко прототипувати і налаштовувати елементи інтерфейсу, такі як кнопки, панелі здоров’я, індикатори боєприпасів та інші важливі елементи. Завдяки такому підходу вдалося забезпечити інтуїтивно зрозумілий і зручний інтерфейс, який інтегрується безпосередньо з ігровою механікою та дозволяє гравцеві без зайвих зусиль взаємодіяти з елементами гри.

2.3 Розробка моделі системи

Для кращого розуміння логіки функціонування ігрової механіки взаємодії зі зброєю в межах розробленого проєкту було прийнято рішення створити модель роботи системи у вигляді UML-діаграми діяльності (рис. 2.8). Цей підхід дозволяє наочно продемонструвати етапи взаємодії користувача з системою, внутрішню логіку обробки дій, а також послідовність подій, що відбуваються в грі при використанні зброї.

Згідно з представленою діаграмою, початкова взаємодія починається з натискання кнопки вибору слоту зброї. Система перевіряє, чи є доступна зброя в обраному слоті. Якщо зброя присутня, вона автоматично екіпірується персонажем, що відображається як у функціональному, так і у візуальному аспектах гри (наприклад, модель зброї з’являється в руках персонажа).

Після успішного екіпірування користувач отримує можливість натиснути кнопку пострілу. У цей момент система перевіряє наявність набоїв у магазині. Якщо боєприпаси є, активується процес створення пострілу: відтворюються візуальні ефекти (спалах, дим, мука віддачі), звукові ефекти (звук пострілу), а також ініціюється запуск фізичної кулі або трасера. Ці елементи мають власну фізику руху, яка реалізована через спеціалізовані компоненти рушія Unreal Engine, зокрема Projectile Movement Component.

Коли куля або трасер рухається в просторі гри, система відстежує наявність перешкод. У разі зіткнення відбувається виклик спеціального методу обробки попадання, який аналізує кілька параметрів: тип і щільність матеріалу, кут попадання, початкову швидкість кулі та інші. Залежно від цих даних система приймає рішення про подальшу поведінку кулі: вона може рикошетити від поверхні (зміна напряму польоту та зниження швидкості), пробити об’єкт наскрізь (із можливістю враження наступної цілі) або зупинитися, наносячи шкоду об'єкту

Кожен із результатів обробки супроводжується відповідними ефектами: наприклад, при влучанні – іскри, плями крові, частки уламків, звукові ефекти ураження, вібрації тощо. Це сприяє глибшому зануренню гравця у гру та забезпечує відчуття реалізму.

З метою кращої візуалізації та розуміння внутрішньої логіки модуля взаємодії зі зброєю, діаграма діяльності системи була наведена на рисунку 2.3, що дозволяє представити роботу ключових функціональних блоків у структурованій формі. Такий підхід сприяє ефективному плануванню, тестуванню та подальшій модернізації системи.

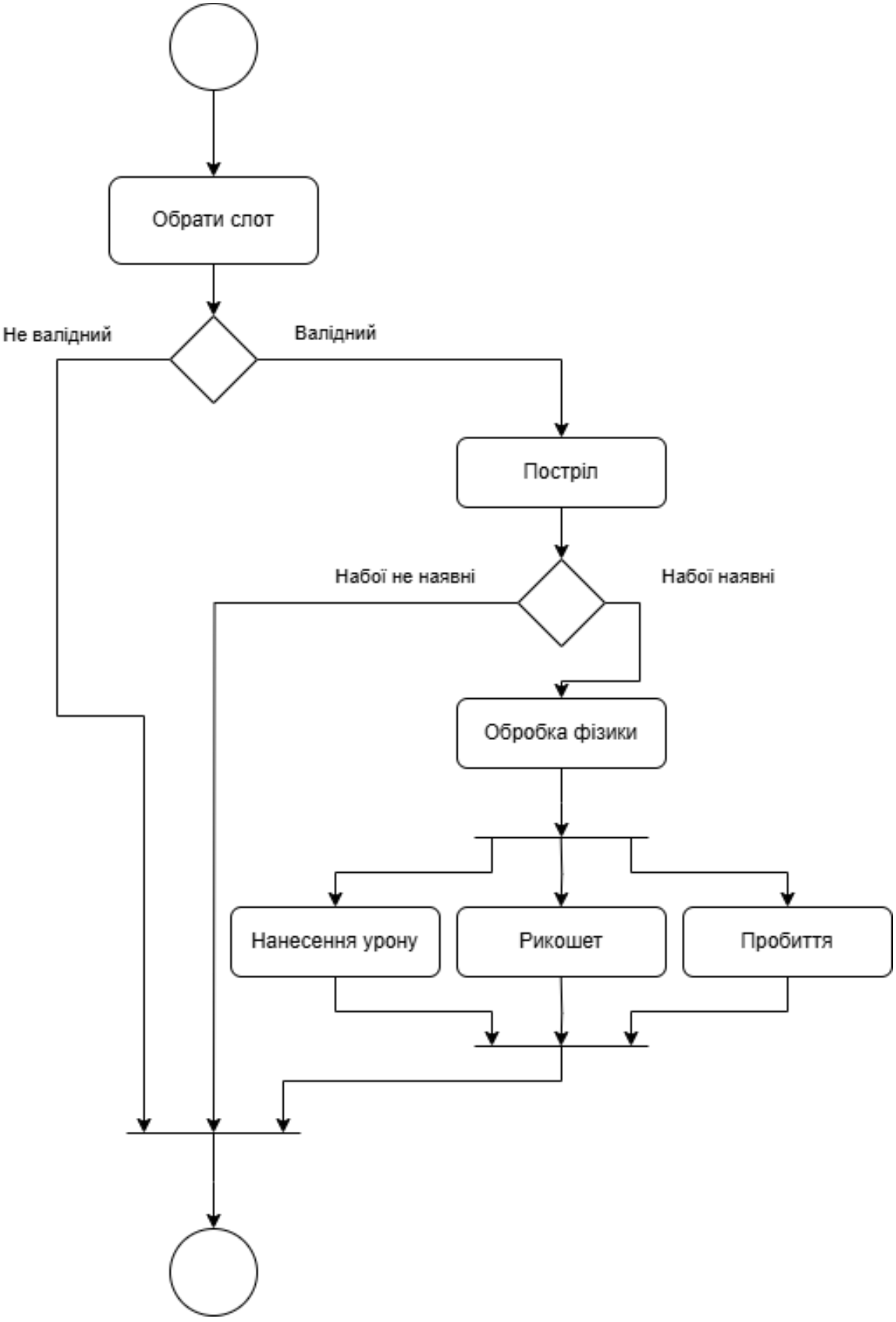

Рисунок 2.3 – Модель програми у вигляді діаграми послідовностей

Діаграму було розроблено у веб-застосунку Drawio.

2.4 Розробка алгоритмів роботи системи

Для реалізації True First Person гри Rock In A Hard Place: Lonesome Road було сформовано загальний алгоритм, який визначає логіку роботи основної бойової механіки. Цей алгоритм включає в себе усі ключові етапи взаємодії гравця зі зброєю та процесом пострілу, починаючи від екіпірування до обробки результатів попадання кулі. Детальний опис алгоритму дозволяє чітко структурувати логіку гри та забезпечити узгоджену взаємодію між різними модулями системи (див. рисунок 2.4).

На першому етапі гравець за допомогою модуля екіпірування обирає активний слот, у якому знаходиться зброя. Система перевіряє наявність доступної зброї у вибраному слоті, після чого відбувається її екіпірування. Це забезпечує готовність персонажа до ведення вогню.

Коли гравець натискає кнопку пострілу, відбувається виклик модуля зброї, який відповідає за логіку використання конкретного виду озброєння. У першу чергу модуль перевіряє кількість набоїв у магазині. Якщо боєприпасів немає, гравцю виводиться відповідне повідомлення (наприклад, звуковий сигнал або текстове повідомлення про відсутність набоїв), і виконання подальших дій припиняється.

Якщо ж набої наявні, система зменшує їх кількість на одиницю та ініціює створення об’єкта кулі, який передається до модуля кулі. Цей модуль відповідає за фізичну симуляцію польоту кулі та її взаємодію з ігровим світом. Траєкторія кулі обчислюється з урахуванням параметрів пострілу. Саме через True First Person куля вилітає саме зі стволу зброї, навідміну від більшості ігор в яких вона створюється усередині віртуальної камери, та вистріл зі зброї є лише візуальним ефектом.

Після запуску кулі система постійно відслідковує її переміщення та визначає можливість зіткнення з об’єктами навколишнього середовища або супротивниками. Обробка зіткнення відбувається в залежності від типу кулі, матеріалу об’єкта, кута влучення та наявності броні.

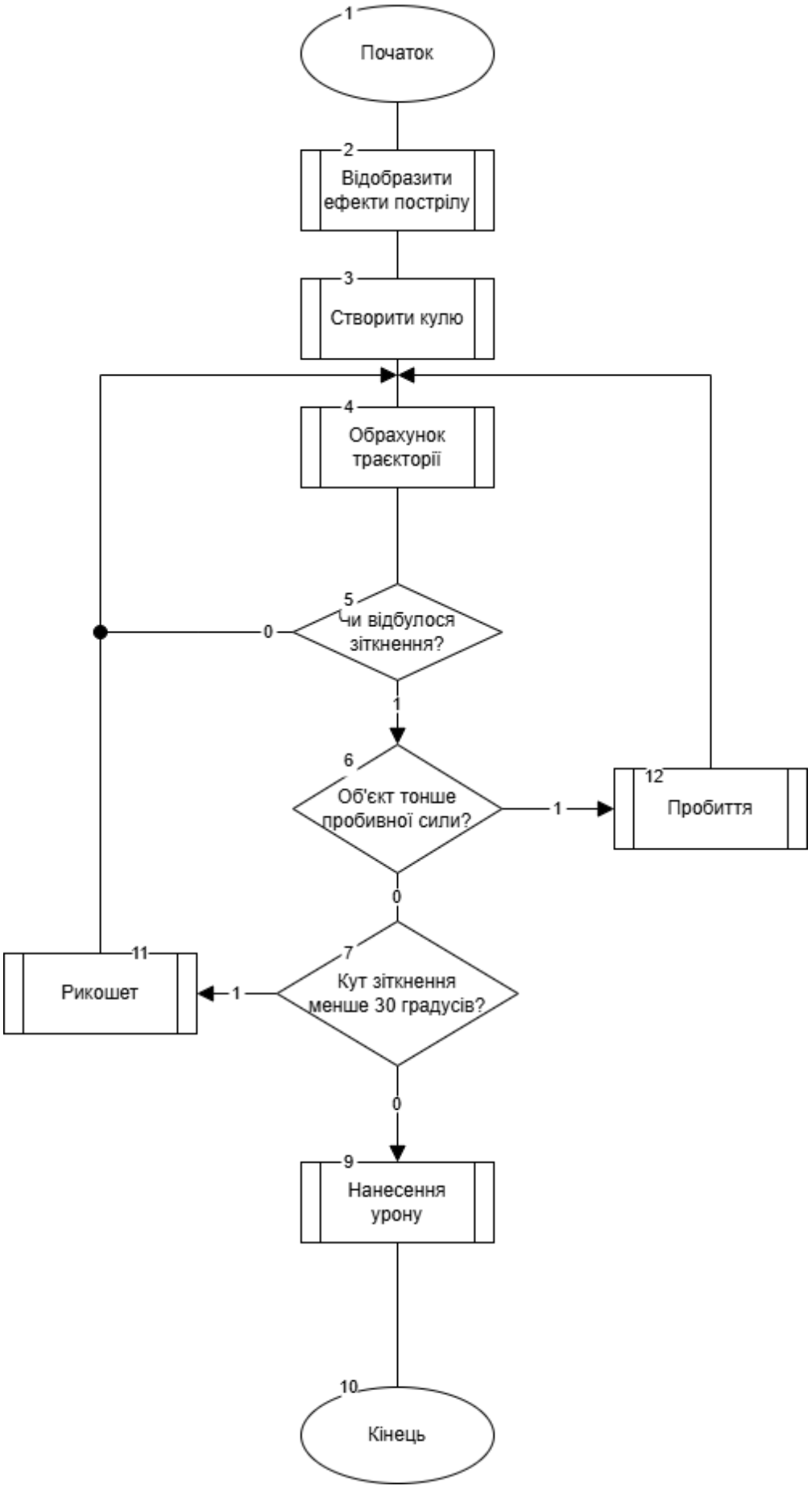

Рисунок 2.4 – Алгоритм роботи модулю кулі

2.5 Висновки

У другому розділі було розглянуто процес створення графічного інтерфейсу користувача для шутер-гри *Rock In A Hard Place: Lonesome Road*. Також було розроблено загальний алгоритм роботи програмного застосунку, що описує взаємодію між модулями екіпірування, зброї та кулі під час бойових дій. Окрему увагу було приділено модулю кулі, в якому реалізовано обчислення траєкторії польоту, логіку взаємодії з об’єктами оточення, а також обробку зіткнень і нанесення ушкоджень

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації програмного забезпечення

Для розробки шутер-гри Rock In A Hard Place: Lonesome Road було обрано низку сучасних технологій, що забезпечують ефективну реалізацію ігрової механіки, високий рівень візуалізації та зручність розробки.

Unreal Engine 5 було обрано як основне середовище розробки завдяки його потужному рушію, який надає широкі можливості для створення фотореалістичної графіки, роботи з фізикою, анімаціями та штучним інтелектом. Unreal Engine 5 забезпечує гнучку архітектуру, що дозволяє ефективно реалізовувати складні механіки стрільби, управління персонажем та інтерактивну взаємодію з навколишнім середовищем.

У порівнянні з іншими рушіями, такими як Godot чи Unity, Unreal Engine 5 має значну перевагу в плані графічної потужності та готових інструментів для створення високоякісного 3D-контенту. Наприклад, Unity більше підходить для мобільних та 2D-ігор, тоді як Godot орієнтований на інді-проєкти з обмеженим бюджетом і має менше вбудованих рішень для складної 3D-фізики. Unreal Engine також має підтримку Nanite і Lumen — технологій, що дозволяють досягти високого рівня деталізації та реалістичного освітлення без значного зниження продуктивності.

Мова програмування C++ була обрана як основна для написання ігрової логіки, оскільки вона є рідною для Unreal Engine і дозволяє отримати максимальну продуктивність та контроль над виконанням коду. У порівнянні з C# (який використовується у Unity), C++ вимагає більше уваги до управління пам’яттю, але водночас забезпечує вищу гнучкість і кращу оптимізацію на низькому рівні, що критично важливо для реалізації складних шутер-механік.

Visual Studio виступає як інтегроване середовище розробки (IDE), що забезпечує зручне програмування на C++ у поєднанні з Unreal Engine. У

порівнянні з іншими IDE, такими як Rider або Code::Blocks, Visual Studio пропонує найкращу інтеграцію з Unreal Engine, має потужну систему автодоповнення, розвинений налагоджувач і інструменти аналізу продуктивності, що значно спрощують розробку великих проєктів.

Для реалізації фізики кулі та її руху було обрано стандартний компонент Unreal Engine — Projectile Movement Component. Цей компонент забезпечує повністю інтегровану систему для моделювання руху снарядів із підтримкою гравітації, опору повітря та зіткнень з об’єктами середовища. У порівнянні з ручною реалізацією фізики через Tick-функції чи кастомні компоненти, використання Projectile Movement Component значно спрощує розробку поведінки кулі, забезпечує вищу стабільність та дозволяє швидше досягти бажаного результату.

Отже, вибрані інструменти — Unreal Engine 5, C++, Visual Studio та стандартні компоненти UE5 — дозволяють створити технічно досконалу, візуально вражаючу гру з максимально ефективним використанням часу і ресурсів розробки.

3.2 Розробка модуля екіпіювання

При розробці модуля екіпірування для шутер-гри Rock In A Hard Place: Lonesome Road важливо враховувати специфіку взаємодії гравця зі зброєю. Модуль екіпірування відповідає за вибір та керування слотами зброї, що дозволяє гравцю оперативно змінювати активну зброю під час гри. Через інтерфейс модуля екіпірування гравець має змогу призначати різні види зброї до визначених слотів, активувати необхідний слот у будь-який момент гри, а також взаємодіяти з іншими елементами екіпірування. Такий підхід забезпечує гнучкість та динамічність бойового процесу, а також створює передумови для реалізації розширеної механіки інвентарю та кастомізації зброї. У ньому міститься мапа слотів, та самі слоти. Цей метод також візуально відображає зброю на ігровому персонажі, це в сукупності з true first person поглядом надає

можливість гравцю візуально бачити своє екіпіювання та зброю на собі. Модуль є компонентом персонажа. Модуль зображено на рисунку 3.1.

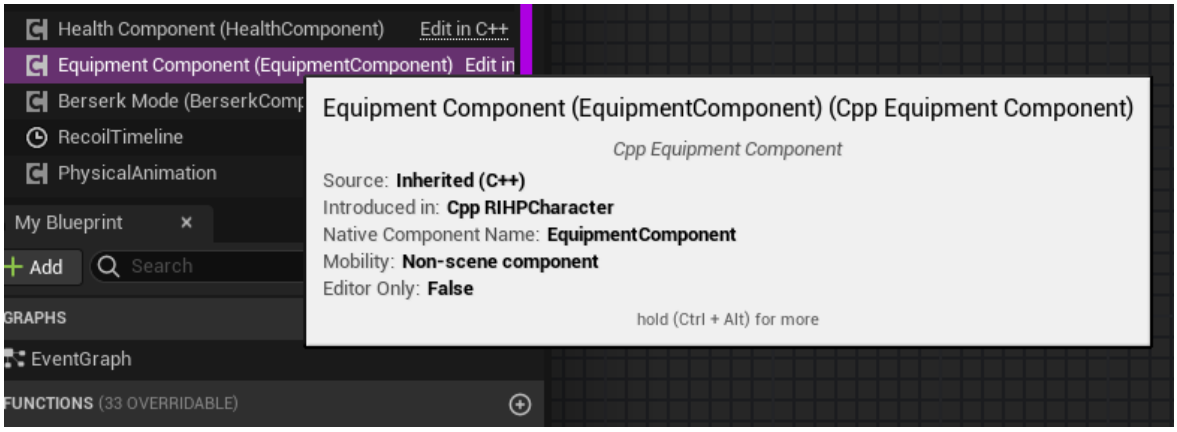

Рисунок 3.1 – Модуль екіпіювання

Гравець має змогу обирати слот, ця дія використовує метод діставання зброї. Модуль екіпірування також виконує перевірку на доступність обраного слоту зброї. Перед активацією зброї, програма перевіряє, чи призначена до відповідного слоту конкретна одиниця зброї та чи не є слот порожнім. У випадку, якщо слот є недоступним або незаповненим, гравцю виводиться повідомлення про неможливість використання зброї. Лише після успішної перевірки доступності слоту гравець отримує можливість здійснити постріл, ініціюючи відповідну послідовність дій у модулі зброї та модулі кулі.

У таблиці 3.1 зображено доступні гравцю слоти зброї

Таблиця 3.1 – таблиця слотів зброї

Слот 1	Acpp_ItemWeapon* PrimaryWeaponSlot;
Слот 2	Acpp_ItemWeapon* SecondaryWeaponSlot;
Слот 3	Acpp_ItemWeapon* SidearmWeaponSlot;

Слоти є різними за можливістю розміщення зброї залежно від класу, слот 1 та 2 можуть містити повнорозмірну зброю, в той час як слот 3 може містити зброю тільки пістолетного типу.

У процесі розробки бойової механіки Rock In A Hard Place: Lonesome Road було реалізовано модуль системи віддачі зброї. Для цього були створені два методи — RecoilProgressVertical та RecoilProgressHorizontal.

Метод RecoilProgressVertical відповідає за вертикальну складову віддачі. Після здійснення пострілу він викликається для імітації підкидання ствола вгору, що змушує гравця коригувати прицілювання після кожного пострілу. Цей ефект реалізовано шляхом поступового зміщення камери або напрямку прицілу на певний кут, що залежить від типу зброї, кількості пострілів та режиму стрільби.

Метод RecoilProgressHorizontal реалізує горизонтальну (бокову) складову віддачі, яка впливає на непередбачуваність траєкторії пострілів під час черги. Горизонтальна віддача зазвичай реалізується як випадкове або напіввипадкове зміщення камери вліво або вправо, що створює потребу у додатковому контролі з боку гравця.

Разом ці два методи формують динамічну модель віддачі, яка враховує тип зброї, режим вогню (одиночний/автоматичний), темп стрільби та попередню віддачу. Їх використання дозволяє створити складну, але передбачувану поведінку зброї, що мотивує гравця до освоєння та тренування навичок стрільби, підвищуючи загальну глибину ігрового процесу.

У межах реалізації системи стрільби гри Rock In A Hard Place: Lonesome Road було розроблено метод, який відповідає за отримання параметрів віддачі від активної зброї та їх інтеграцію в модуль екіпірування. Основною функцією цього методу є отримання даних про поведінку віддачі конкретного зразка зброї та прив’язка відповідної кривої віддачі (recoil curve) до системи екіпірування персонажа.

Після того як користувач екіпірує зброю, метод зчитує з неї набір характеристик, які включають: тип кривої вертикальної віддачі, амплітуду горизонтального зміщення, силу віддачі та швидкість її затухання. Ці параметри

зберігаються у структурі даних зброї та можуть бути індивідуальними для кожного виду озброєння.

Після зчитування, ці параметри прив’язуються (bind) до відповідної анімаційної або математичної кривої в системі екіпірування, що дозволяє симулювати віддачу безпосередньо в момент пострілу. Таким чином, кожна зброя в грі має унікальну модель поведінки, що відображає її габарити, калібр, вагу та інші технічні особливості.

Цей підхід забезпечує гнучкість системи — нові зразки зброї можна додавати без необхідності змінювати основну логіку віддачі, достатньо лише налаштувати відповідні криві та передати їх через цей метод. У результаті гравець отримує реалістичну та варіативну стрільбу, що є ключовим елементом бойової механіки.

Криві віддачі задаються в двох окремих асетах, їх приклади зображено на рисунках 3.2 та 3.3.

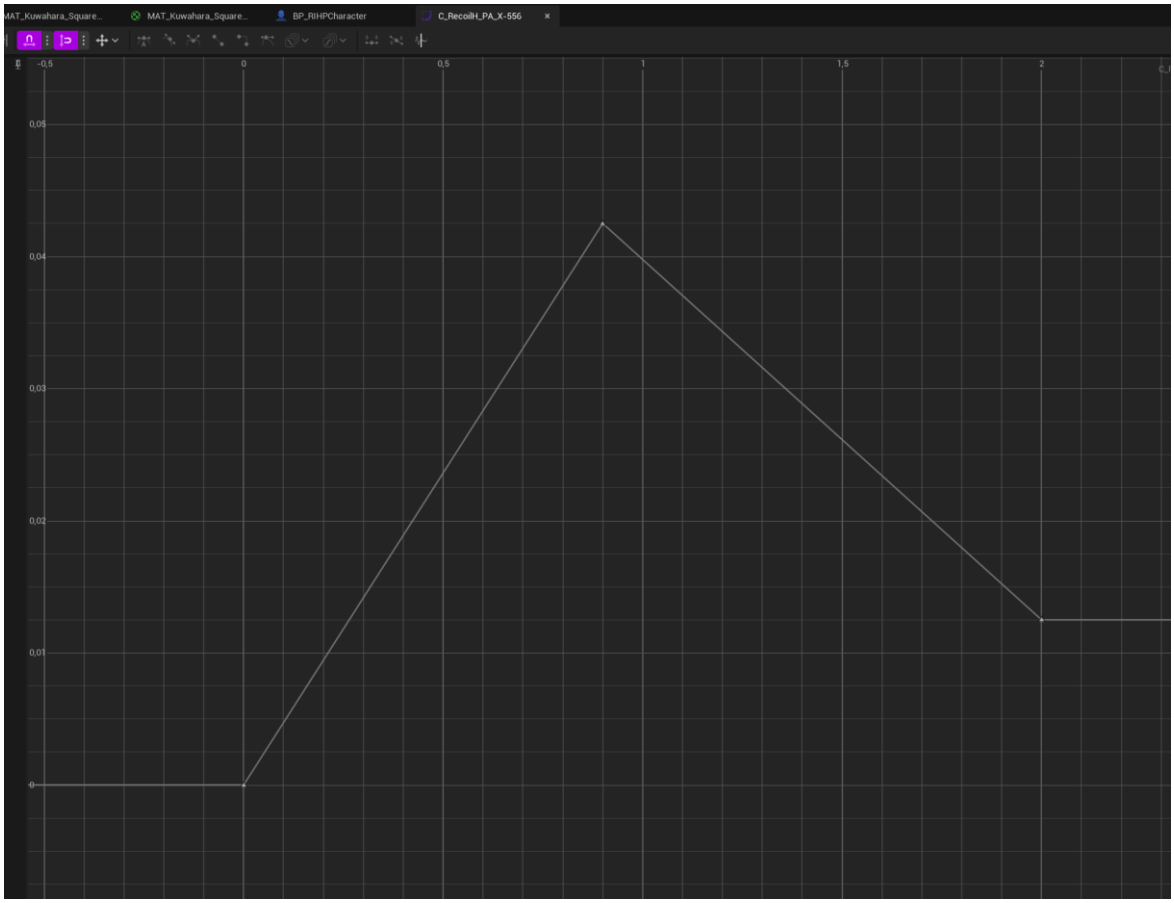

Рисунок 3.2 – Крива горизонтальної віддачі

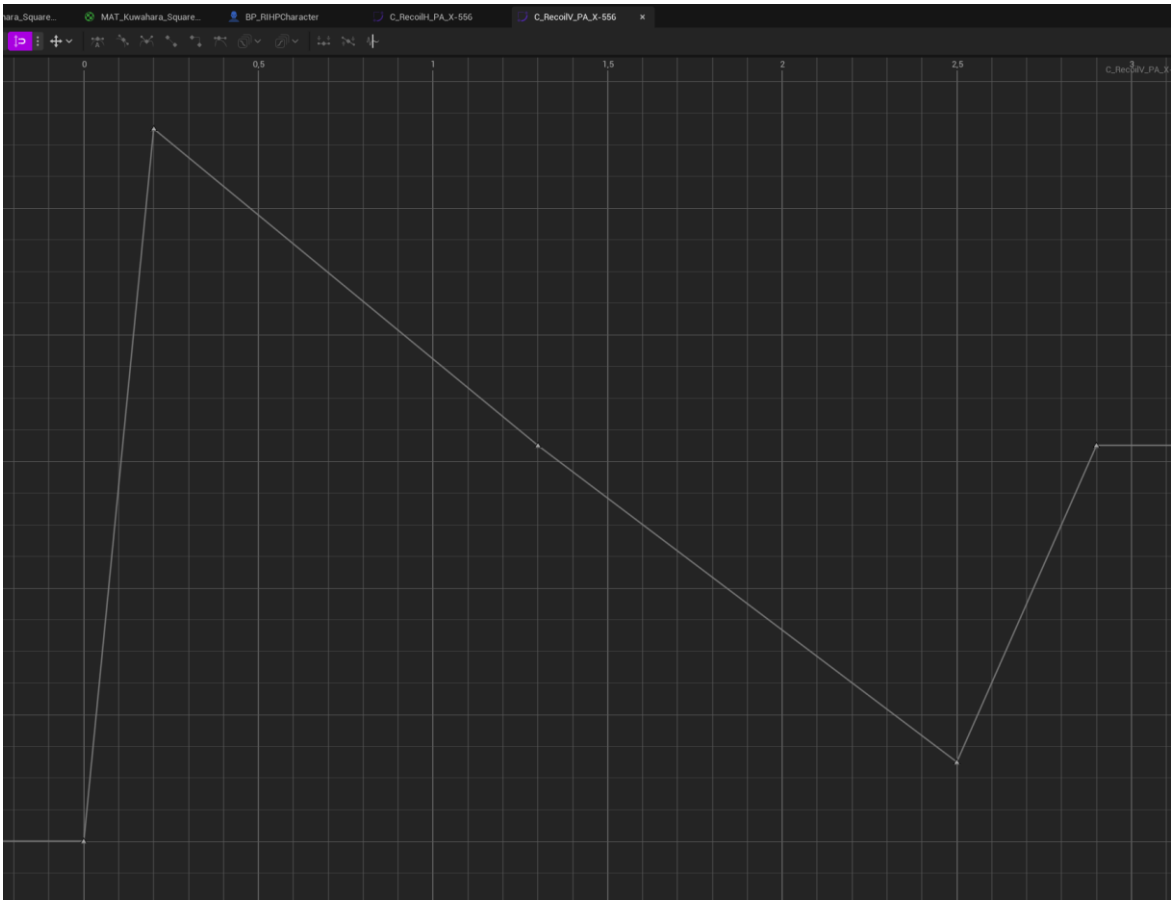

Рисунок 3.3 – Крива вертикальної віддачі

Криві зберігаються як окремі файли, та динамічно завантажуються підчас ігрового процесу. На рисунку 3.4 зображено приклад цих файлів.

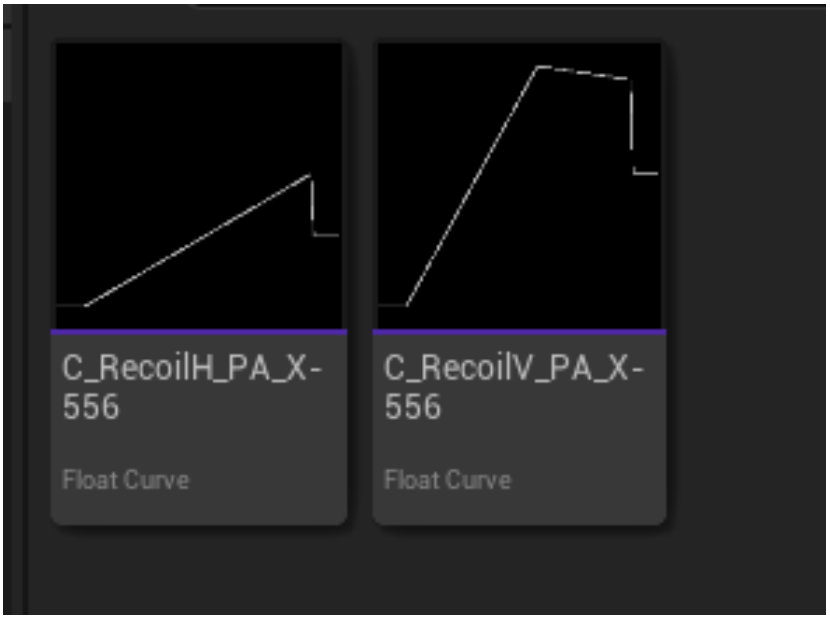

Рисунок 3.4 – Приклад файлів віддачі

При стрільбі ці криві зчитуються, при кожному пострілі повзунок програвання зсовується на 0,1 поділку, вертикальне значення кривої вводиться в систему та зміщує приціл гравця на зазначене число. Підчас цього записується дельта зміщення.

Було розроблено метод CompensateRecoil, який відповідає за плавне повернення зброї до початкової позиції після завершення стрільби. Метод використовує дельту зміщення записану минулим методом для компенсації віддачі. Після завершення стрільби автоматично спрацьовує встановлена затримка, після заверешення якої спрацьовує модуль, він плавно повертає приціл гравця на початкову позицію, віднімаючи змінну зміщення від позиції прицілу у даний час.

3.3 Розробка модуля кулі

Куля реалізована як actor, об’єкт що має змогу бути розміщеним у світі. Вона складається з візуальної частини (mesh), коробки колізії (collision box), логіки руху (projectile movement Візуальна частина існує тільки для гравця, вона не впливає на рух кулі, тільки слугує для її відображення. Коробка колізії використовується для оброки взаємодії кулі з довколишнів світом, вона має всього 4 полігони, що є оптимальним для обробки зіткнень, будь-яка інша геометрична форма з більшою кількістю полігонів створювала б додаткове навантаження на процесор для обробки, та ускладнювала створення декількох одночасних пострілів. Логіка руху є встановленим компонентом у рушії та оптимізована для подібного роду об’єктів. Вона зміщує кулю враховуючи встановлену швидкість. Візуальний вигляд кулі зображено на рисунку 3.5.

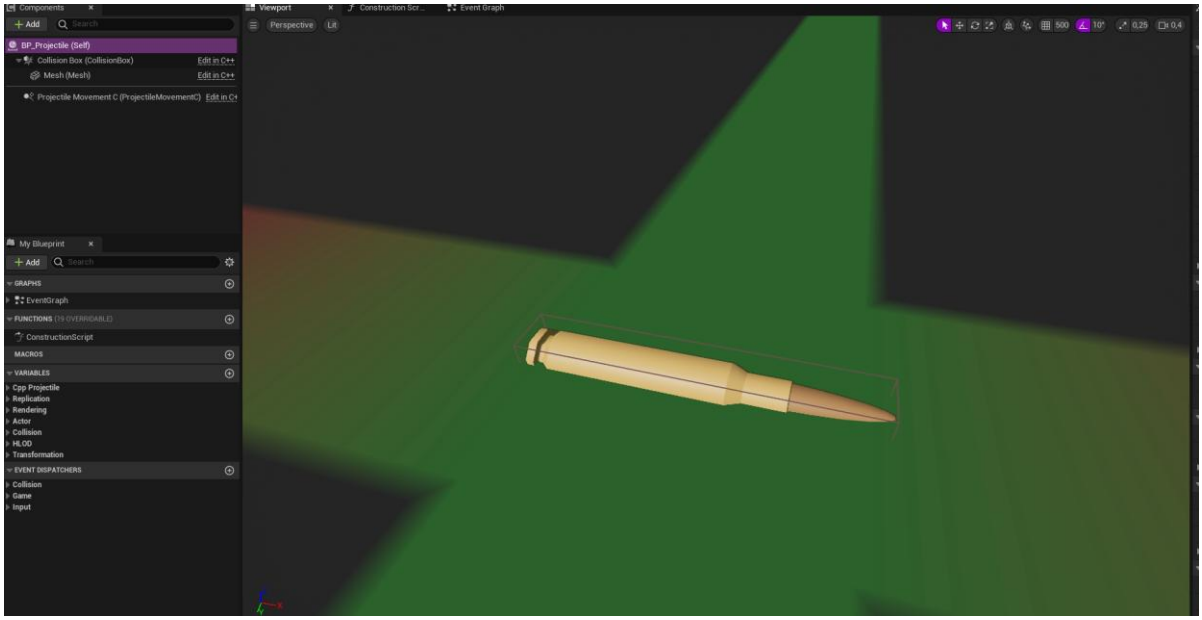

Рисунок 3.3 – Куля

До візуальної частини входить система Niagara, вона відображає трасер кулі, коли та рухається у просторі. Трасер зображено на рисунку 3.4.

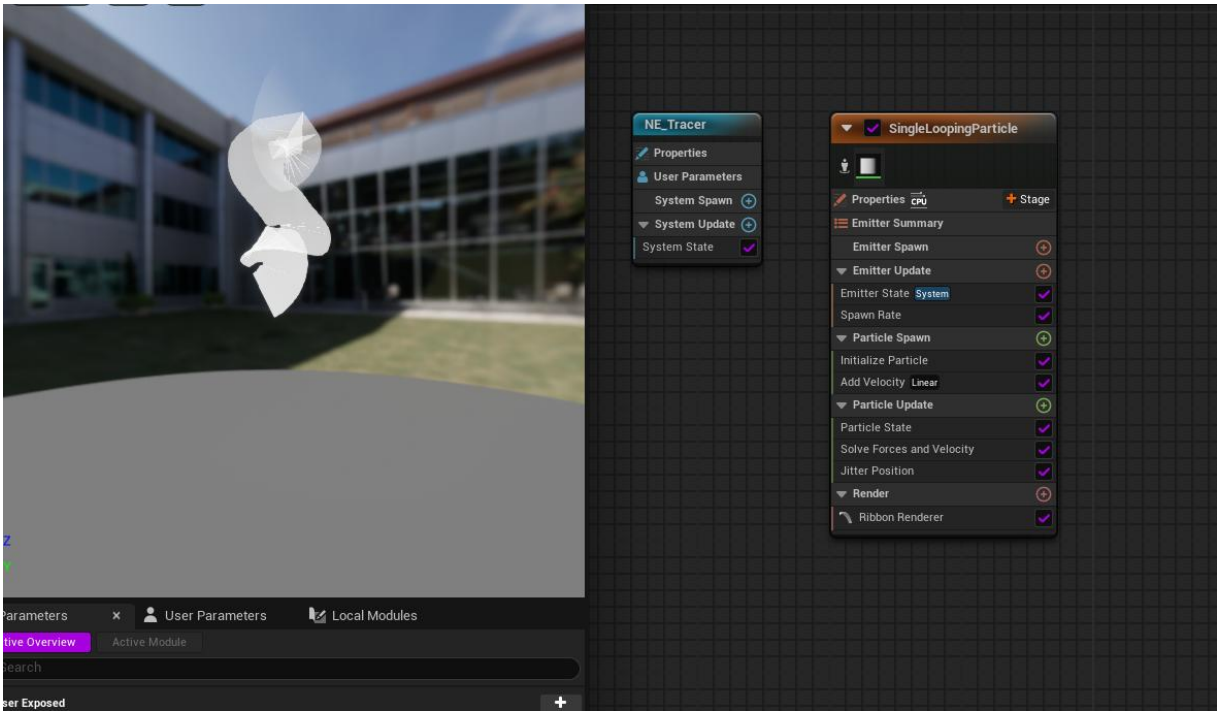

Рисунок 3.3 – Система Niagara для створеня ефекту трасеру

Для коректної роботи кулі у грі було розроблено спеціальний метод, який призначає їй початкову швидкість у момент створення. Цей метод викликається

безпосередньо під час ініціалізації кулі та задає напрямок і силу руху, що відповідає напрямку прицілу гравця. Задання початкової швидкості є критично важливим для забезпечення реалістичної та передбачуваної фізичної поведінки снаряда, а також для узгодженої взаємодії з компонентом Projectile Movement Component, який опрацьовує подальший рух кулі у просторі.

У процесі розробки гри також було реалізовано метод обробки попадання кулі, який відповідає за аналіз результатів зіткнення з об'єктами в ігровому середовищі. Цей метод виконує обчислення товщини матеріалу, з яким стикнулась куля, та порівнює її з пороговим значенням, необхідним для пробиття. Якщо товщина матеріалу не перевищує допустиму межу, куля продовжує рух, імітуючи пробиття перешкоди. В іншому випадку куля зупиняється або зникає.

Метод рикошету було розроблено для моделювання ситуацій, коли куля не пробиває перешкоду, а відбивається від неї. У разі зіткнення з твердою поверхнею, метод змінює траєкторію руху кулі відповідно до кута відбиття, використовуючи вектори напряму та нормалі поверхні. Окрім зміни напряму, також відбувається зменшення швидкості кулі, що дозволяє імітувати втрату енергії під час рикошету. Це забезпечує більш реалістичну та динамічну поведінку снарядів у грі.

3.4 Висновки

У третьому розділі було обґрунтовано вибір технологій та інструментів, що використовувались для розробки шутер-гри Rock In A Hard Place: Lonesome Road*. Для створення гри було обрано сучасне ігрове середовище Unreal Engine 5, яке забезпечує потужні можливості для розробки реалістичної графіки, фізики, анімацій та взаємодії з ігровим світом. Мову програмування C++ було обрано як основну завдяки її високій продуктивності та тісній інтеграції з Unreal Engine. Інтегроване середовище розробки Visual Studio використано для зручного написання коду, налагодження та управління проєктом.

Для реалізації фізики руху кулі було використано компонент Projectile Movement Component, який надає готову систему для моделювання траєкторії снарядів з урахуванням фізичних параметрів, таких як гравітація та зіткнення. У розділі також було детально описано розробку відповідних програмних модулів, зокрема модуля екіпірування, модуля зброї та модуля кулі, які забезпечують логіку вибору зброї, перевірки доступності слоту, здійснення пострілу та обробки фізичної поведінки кулі у просторі.

4 ТЕСТУВАННЯ ПРОГРАМИ

4.1 Тестування системи

Тестування програмного забезпечення – це важливий процес, спрямований на перевірку функціональності, надійності, безпеки та сумісності програмного продукту. Метою тестування є виявлення дефектів або помилок у програмному забезпеченні, щоб забезпечити його відповідність вимогам замовника та гарантувати стабільність і коректну роботу в різних умовах експлуатації. Тестування проводиться на різних етапах розробки та включає різноманітні методи, від функціональних тестів до тестів на продуктивність, безпеку і сумісність.

Тестування гри, на відміну від тестування звичайного програмного продукту, має свої особливості, зокрема через інтерактивний характер гри, де важливо забезпечити не лише технічну стабільність, але й комфорт користувача, графічні ефекти, взаємодію з ігровим середовищем та реакцію на дії гравця. Тестування гри зазвичай охоплює такі аспекти, як перевірка механіки ігри, точність фізичних симуляцій, поведінка штучного інтелекту, а також взаємодія з інтерфейсом та налаштуваннями графіки.

Під час тестування гри важливо враховувати її ігровий процес, наявність багів, можливість зависання чи інших проблем, що можуть виникнути під час взаємодії користувача з ігровим середовищем. Окремо тестуються функції, такі як контроль над персонажем, а також перевірка різних режимів гри, звукових і візуальних ефектів, перевірка працездатності в різних умовах (наприклад, на різних операційних системах або пристроях).

Однією з основних відмінностей тестування гри є необхідність вивчення поведінки системи в умовах високої динаміки. У той час як звичайне програмне забезпечення тестується переважно за допомогою ручних або автоматизованих скриптів, які перевіряють введення та виведення даних, тестування гри включає в себе симуляцію реальних дій користувача, перевірку візуальних ефектів, а

також перевірку фізичних моделей, анімацій та інших аспектів взаємодії в реальному часі.

Першим етапом тестування є перевірка можливості запуску гри на доступній конфігурації. Це включає перевірку сумісності гри з конкретними характеристиками комп'ютера, зокрема процесором, відеокартою, оперативною пам'яттю та операційною системою. На цьому етапі тестується, чи відповідає система мінімальним вимогам гри, чи можна запустити гру без помилок, а також визначається, чи стабільно працює вона на даній конфігурації комп'ютера. Якщо гра не запускається або має проблеми при старті, виводяться відповідні повідомлення про помилки, які можуть вказувати на недостатню потужність системи або несумісність з певними компонентами. Це дозволяє виявити та усунути основні проблеми, які можуть перешкоджати нормальній роботі гри.

На рисунку 4.1 зображено скріншот роботи гри, на ньому видно відсутність вузуальних артефаків, що свідчить про коректну роботу візуальної частини гри на даній конфігурації.

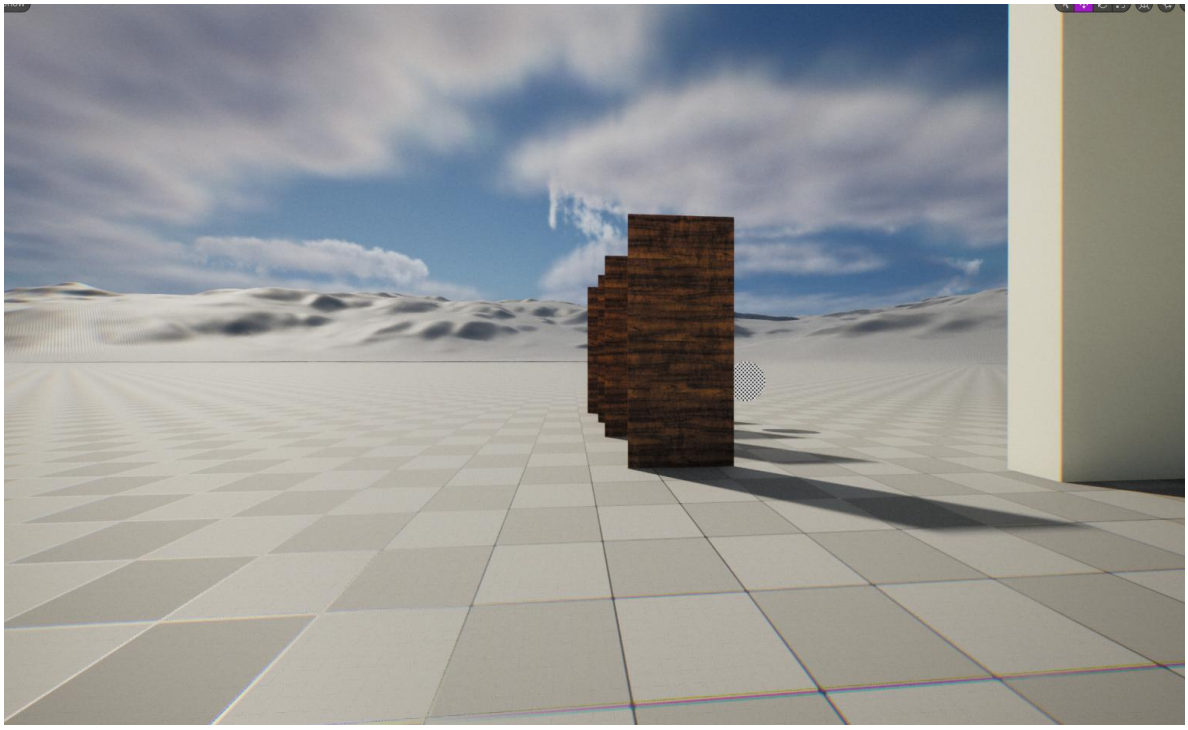

Рисунок 4.1 – Скріншот гри

Наступним етапом тестування є задання базових параметрів зброї, що зображено на рисунку 4.2. На цьому етапі перевіряються основні характеристики кожного виду зброї, такі як швидкість кулі, сила віддачі, точність пострілу, а також ефекти, пов'язані з її використанням (наприклад, звук, візуальні ефекти, рикошети тощо). Всі ці параметри важливі для забезпечення реалістичної поведінки зброї в ігровому середовищі. Тестування цього етапу дозволяє виявити можливі неточності або невідповідності в налаштуваннях зброї, що можуть негативно вплинути на баланс гри або взаємодію з іншими елементами ігрової механіки.

На рисунку 4.2 зображено задану зброю для тестування.

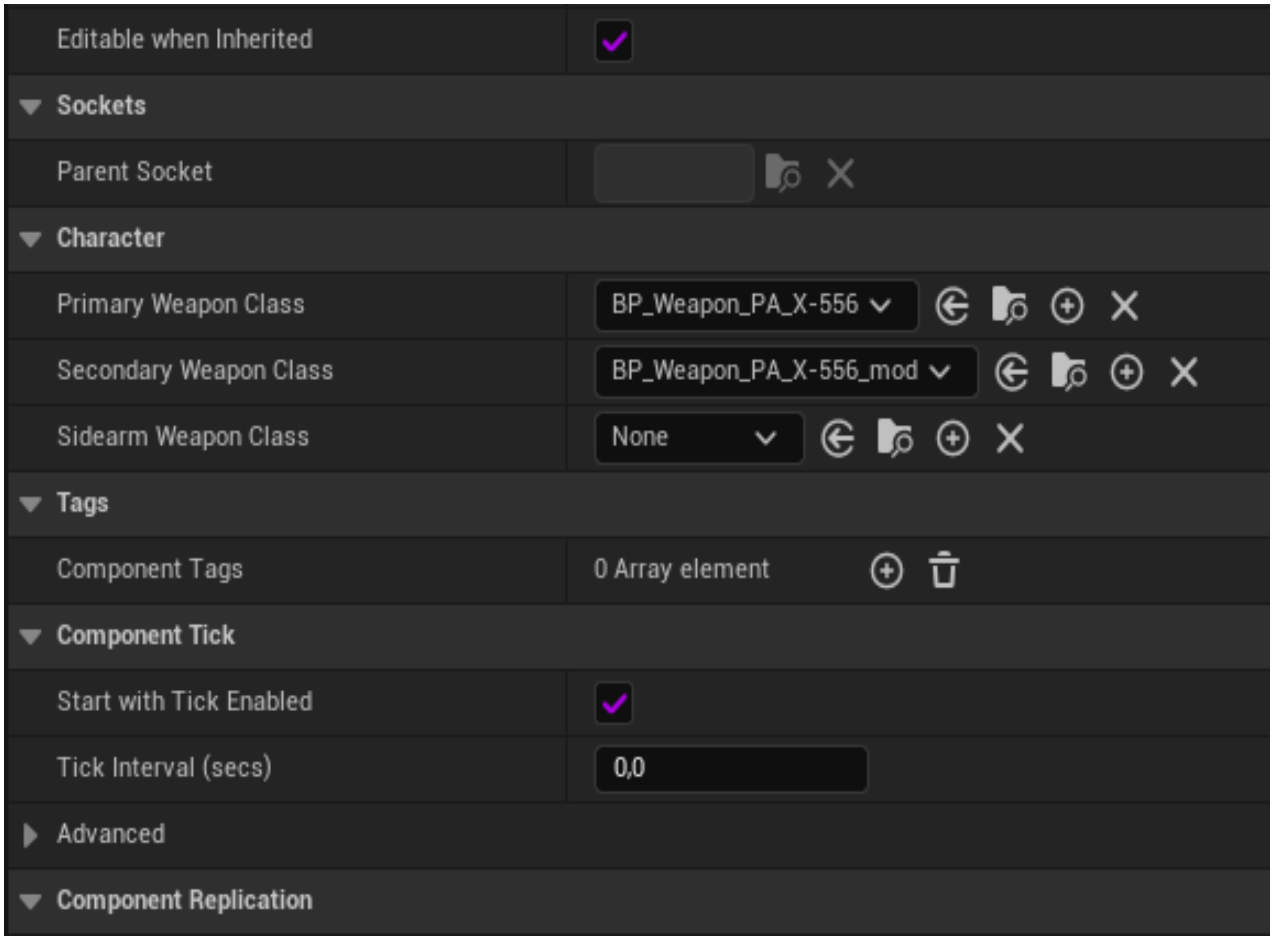

Рисунок 4.2 – Задана зброя

При коректній роботі гри зброя повинна з’явитись на моделі персонажу при запускі. На рисунку 4.3 зображено коректну роботу даного функціоналу.

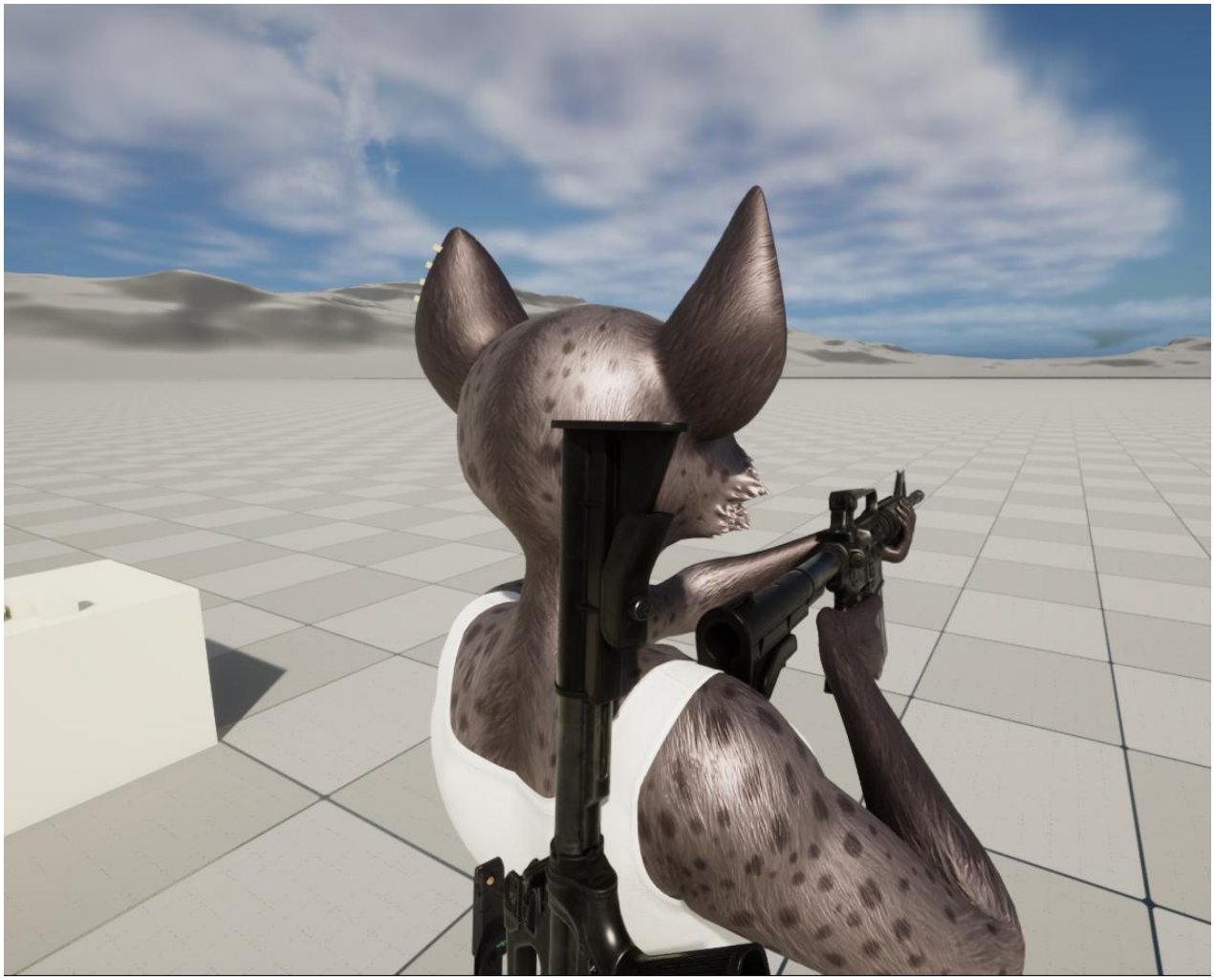

Рисунок 4.3 – Зброя на моделі персонажу

Після Наступним етапом є тестування балістики на створеному тестовому стенді, який складається з декількох шарів перешкод різної товщини. Це тестування дозволяє перевірити, як зброя та її боєприпаси взаємодіють з різними матеріалами, а також як кулі взаємодіють із перешкодами. На цьому етапі перевіряються такі характеристики, як проникність кулі через матеріали, можливість рикошету або зупинки, а також точність визначення того, який саме матеріал може зупинити кулю або, навпаки, дозволити їй пройти через декілька шарів. Тестування на перешкодах важливе для забезпечення реалістичності бойових ситуацій і дозволяє налаштувати параметри проникнення та ефекти пошкодження, що додають глибину ігровому процесу.

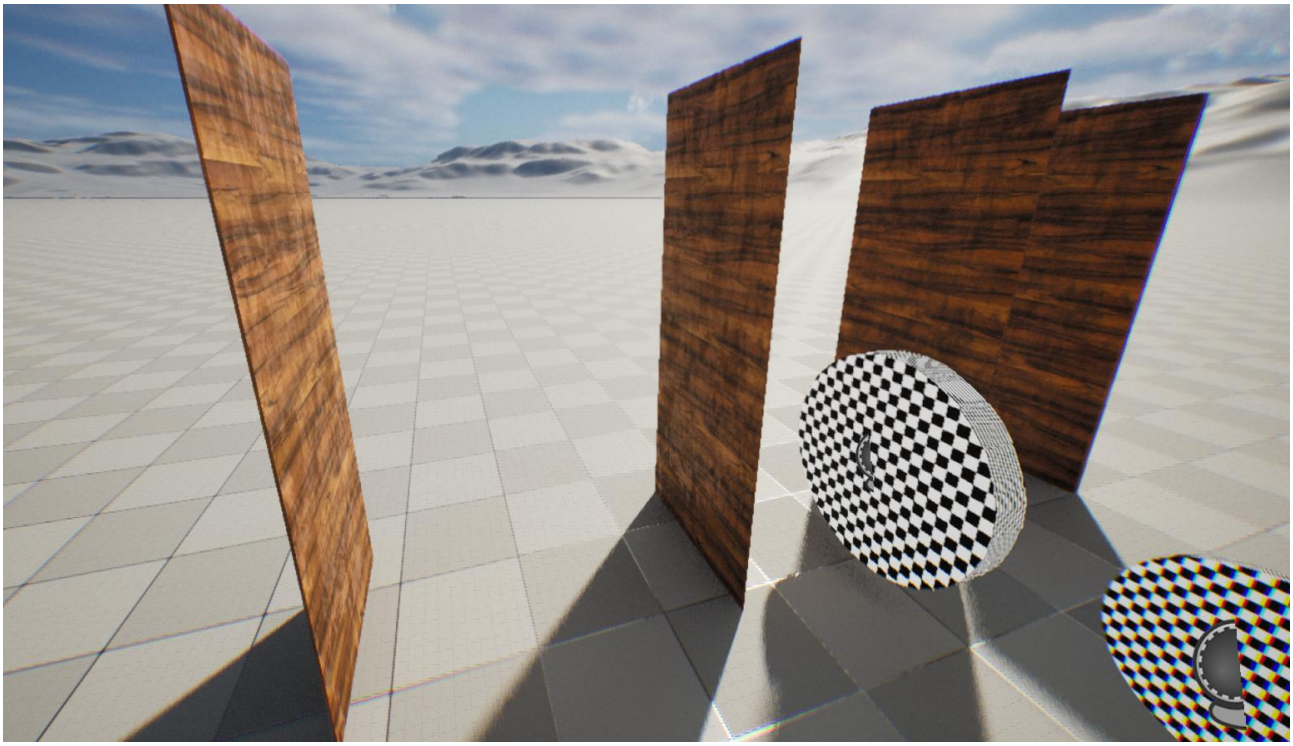

Рисунок 4.4 – Тестовий сенд

При пострілі має відобразитись візуальний ефект, бути створена куля з заданою швидкістю. На рисунку 4.5 зображено коретно оброблений постріл.

Рисунок 4.5 – Тестовий постріл

Далі буде протестовано пробивну силу кулі та її поведінку при взаємодії з поверхнями.

Тестовий стенд створений для забезечення умов для перевірки поведінки балістики куль.

Для першого тесту та перевірки пробивної можливості кулі було виконано постріл у бік тонкої стінки під кутом 90 градусів. При зіткнені куля пройшла наскрізь, що і було необхідним результатом. Результат зображено на рисунку 4.6.

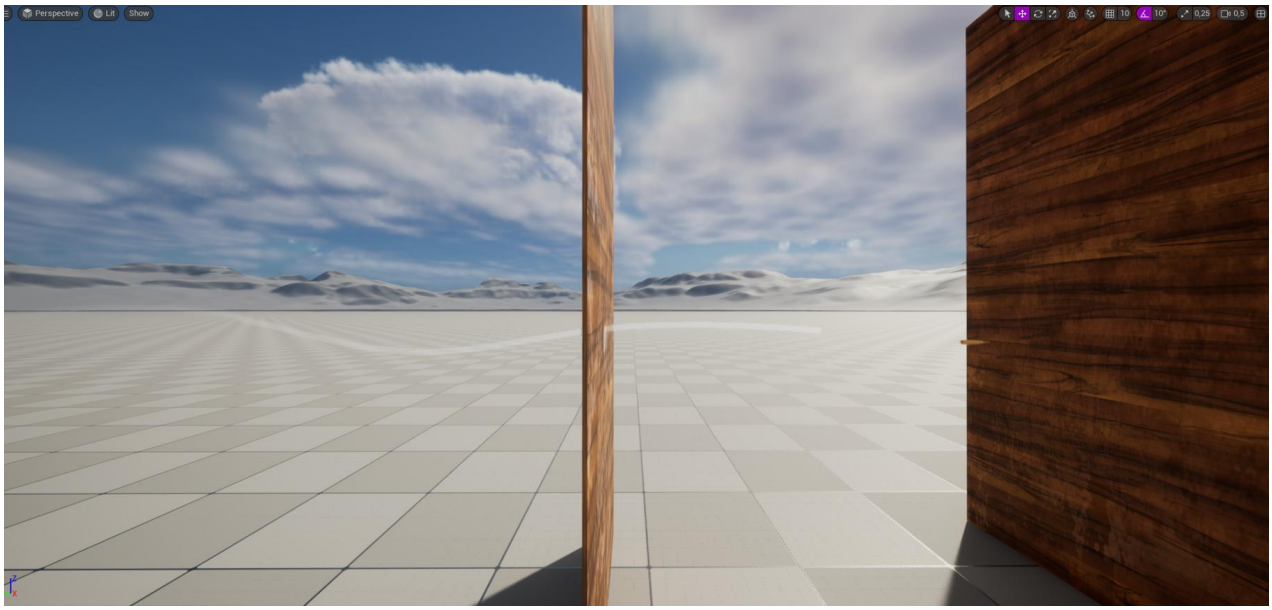

Рисунок 4.6 – Перевірка пробивної сили тонких стінах

З проведених тестів видно, що розроблена система балістики працює коректно: куля проходить крізь тонкі перешкоди (наприклад, дерево або гіпсокартон), поступово втрачаючи швидкість в залежності від товщини та матеріалу об’єкта. При цьому на місці проходу автоматично з’являється декаль пробиття, що візуально підтверджує взаємодію снаряда з поверхнею. У випадку зі значно товстими або міцними перешкодами, куля повністю зупиняється, що свідчить про правильно реалізовану логіку поглинання кінетичної енергії та ефективне використання фізичного трасування.

На рисунку 4.7 зображено повну зупинку купі

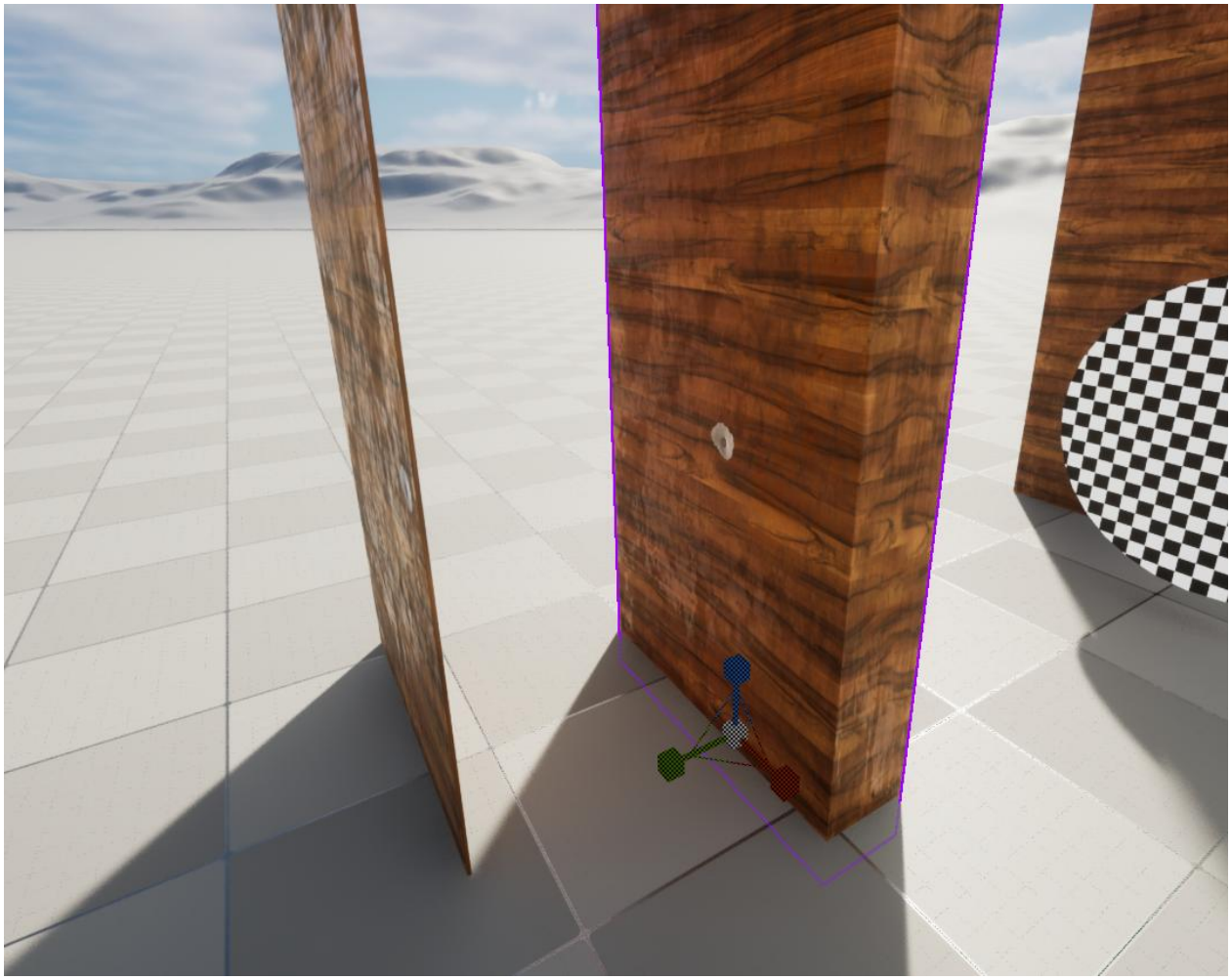

Рисунок 4.7 – Перевірка пробивної сили на товстих стінах

Куля пройшла тонку перешкоду та була повністю зупинена наступною, заливши на ній візуальний ефект.

Наступним важливим етапом у вдосконаленні системи симуляції озброєння є реалізація та перевірка механіки рикошету куль. Це критично важливий компонент для створення реалістичного та глибокого бойового ігрового досвіду, особливо в тактичних шутерах, де важливу роль відіграє фізика взаємодії з навколишнім середовищем.

Рикошет — це явище, при якому куля не проникає в поверхню, а змінює свій напрямок руху після удару. У контексті шутер-ігри це не просто візуальний ефект, а фізично обґрунтована поведінка, яка повинна враховувати низку параметрів:

1. Матеріал поверхні

Різні матеріали мають різну щільність, твердість та енергопоглинаючі властивості. Наприклад, при ударі об бетон або метал куля з більшою ймовірністю рикошетує, особливо під гострим кутом, тоді як тканина, ґрунт або дерево, як правило, поглинають енергію пострілу. Для реалізації цього необхідно впровадити систему матеріальних властивостей поверхонь (наприклад, через Physical Material в Unreal Engine), яка визначає, чи можливий рикошет взагалі і з якою вірогідністю.

2. Характеристики кулі

Вага, швидкість, форма та тип кулі (наприклад, бронебійна, експансивна тощо) також суттєво впливають на здатність до рикошету. Важчі кулі з високою швидкістю можуть проникати глибше або відбиватися під меншим кутом. В моделі симуляції важливо враховувати ці характеристики як частину балістичного профілю кожного типу боєприпасів.

3. Кут удару

Рикошет здебільшого трапляється при гострому куті удару — зазвичай менше 45°. При прямому влучанні (близько 90° до поверхні) куля або проникає, або втрачає всю свою енергію без рикошету. Тому система повинна розраховувати кут між нормаллю поверхні в точці удару та вектором руху кулі. На основі цього визначається, чи відбудеться рикошет, під яким кутом і з якою втратою енергії куля продовжить рух.

Для проведення попереднього тестування механіки рикошету було здійснено постріл кулею під кутом приблизно 30 градусів до поверхні товстого та твердого об’єкта, зокрема бетонної стіни. Такий кут було обрано навмисно, оскільки він є типовим для ситуацій, у яких рикошет є найвірогіднішим — при достатньо гострому ударі, але з достатньою енергією кулі для взаємодії з матеріалом.

У ході тесту було зафіксовано, що куля не проникла крізь поверхню, а натомість відбилася від неї, змінивши напрямок руху у відповідності до очікуваної фізичної моделі. Результати підтверджують, що система коректно

обробляє умови для виникнення рикошету. На рисунку 4.7 зображено рух рикошету.

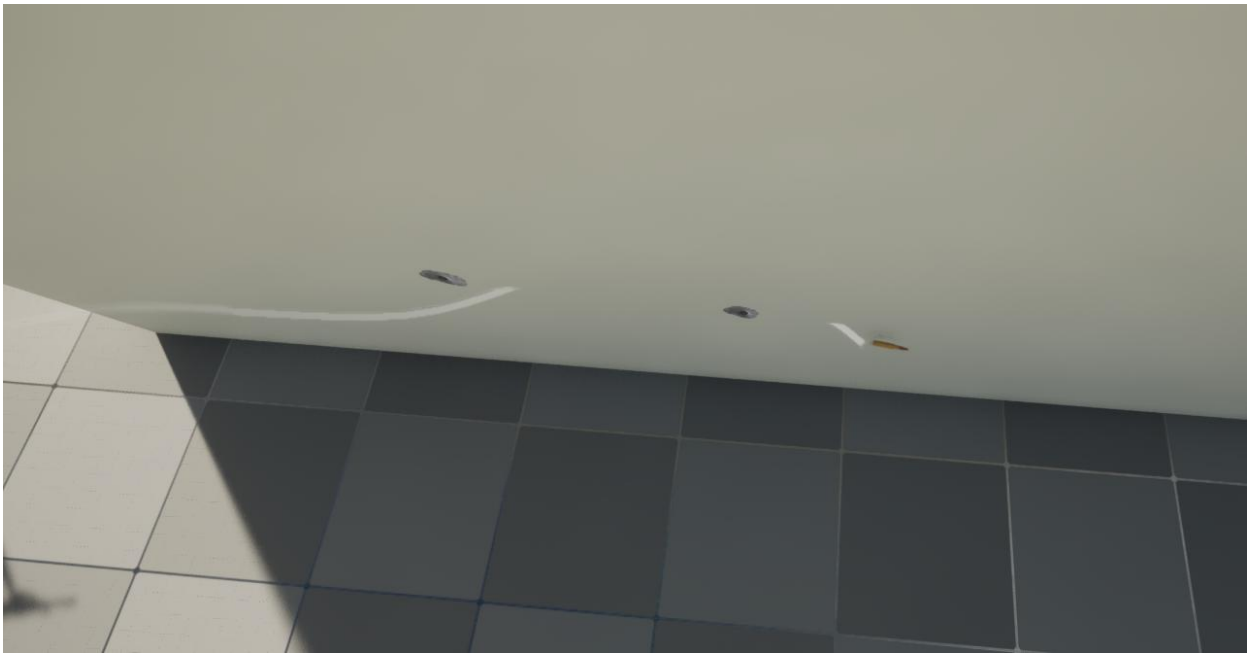

Рисунок 4.7 – Рикошет

У результаті тестування було підтверджено, що після удару об тверду поверхню під кутом приблизно 30 градусів куля змінила свій напрямок руху відповідно до очікуваної траєкторії рикошету. Кут відбивання відповідав фізичним законам, з урахуванням втрати частини кінетичної енергії при зіткненні. Таким чином, тест було успішно пройдено, і система правильно визначила умови для рикошету: характер матеріалу, кут удару та властивості снаряда.

Ця механіка є ключовим елементом реалізму та глибини ігрового процесу, оскільки вона відкриває нові тактичні можливості для гравця. Зокрема, куля, що рикошетує, може досягти цілі, яка перебуває за укриттям або поза прямою лінією вогню, що створює простір для стратегічного використання середовища. Крім того, реалізація рикошету підвищує небезпеку ведення вогню в закритих просторах, де куля може непередбачувано змінити напрямок — це стимулює більш обережну та обдуману гру.

У подальшому планується вдосконалити цю систему, протестувавши різні кути зіткнення, типи поверхонь і боєприпасів, щоб забезпечити максимальну достовірність та варіативність рикошетів у різних умовах.

Наступним етапом тестування стала перевірка механіки розкиду куль, яка є невід’ємною частиною реалізму стрільби в шутерах. Ця система відповідає за випадкове відхилення траєкторії кулі від ідеального вектора прицілювання, імітуючи вплив таких факторів, як неточність зброї, тремтіння рук персонажа, рух під час стрільби, а також режим вогню (одиночний постріл, черга тощо)..

На рисунку 4.8 зображєно стрільбу в автоматичному режимі у прицілювані без руху, ця позиція є стандартною та служить початковою точкою для порівняння з іншими.

Рисунок 4.8 – Стрільба стоячи у прицілюванні

Наступним тестом була стрільба «від бедра», ця стрільба є часто використаною у іграх жанру шутер, але славиться своєю неточністю.

Рисунок 4.9 – Стрільба без прицілювання

У порівнянні зі стрільбою у прицілюванні, видно що точність значно менше, що є бажаним результатом.

Одним із найважливіших елементів стрілецької механіки у сучасних шутерах є система патернів віддачі, яка безпосередньо впливає на геймплей, відчуття стрільби, рівень майстерності гравця та загальний баланс бою. Патерн віддачі — це заздалегідь визначена або напіврандомізована траєкторія, за якою зміщується приціл під час тривалого ведення вогню, особливо чергами або автоматичним режимом. На відміну від простого збільшення розкиду при безперервній стрільбі, патерни дозволяють формувати контрольовану модель поведінки зброї, яку гравець може вивчити, запам’ятати та навчитися компенсувати, що додає глибину та майстерність у стрілецький процес. У таких

іграх, як Counter-Strike: Global Offensive, Apex Legends, Rainbow Six Siege або PUBG, система патернів є визначальним фактором у досягненні високого рівня володіння зброєю. Замість того щоб сподіватися на випадковість, досвідчений гравець знає, як буде поводитися приціл під час стрільби з конкретної зброї, і заздалегідь рухає мишу у зворотному напрямку патерну, компенсуючи підняття ствола та вбік. Це створює важливу навичку, яку можна розвивати, що перетворює стрільбу на процес, подібний до керування інструментом — чим краще гравець знає і відчуває свою зброю, тим точніше він може діяти. З точки зору дизайну гри, патерни віддачі є чудовим способом балансування різних типів зброї: деякі автомати можуть мати вертикальну, просту для компенсації віддачу, але вищий розкид; інші — складнішу, горизонтальну, зі змінами напрямку. Таким чином, кожна зброя має власний «характер», що робить вибір спорядження більш осмисленим, а не просто статистично вигідним. Крім того, система патернів дозволяє створювати «вікна точності», коли перші кілька пострілів особливо точні, а далі ефективність падає — це заохочує короткі черги, зменшує спам автоматичної стрільби та стимулює тактичну поведінку. Із технічної точки зору, реалізація патернів може базуватись як на статичних масивах відхилення, де кожна куля має свою позицію, так і на динамічному обчисленні на основі формули з випадковими або напіввипадковими варіаціями. У більш складних системах можлива адаптація патерну до стану гравця (рух, стрибок, прицілювання, зміна положення), що ще більше ускладнює завдання, але й підвищує реалізм та глибину. Важливо зазначити, що ефективна реалізація патернів віддачі також покращує візуальне та звукове сприйняття стрільби — рух прицілу, поштовх зброї, зміщення камери та звук пострілу повинні працювати в унісон, щоб створити відчуття «живої» зброї, яка має масу, силу та характер. Усе це разом підсилює іммерсивність та задоволення від процесу стрільби, що є центральною частиною більшості FPS-ігор. Нарешті, патерни віддачі — це ще й спосіб надати гравцеві зворотний зв’язок: вони бачать наслідки своїх дій, вчаться контролю, адаптуються до ситуації та з кожною сесією вдосконалюють власні навички. Це сприяє формуванню глибокого

ігрового процесу, в якому навички важать більше за реакцію чи удачу, а також підтримує мотивацію до постійного розвитку та змагання.

На рисунку 4.10 зображено два патерни віддачі, перший був створений стрільбою зі зброї меншого калібру, другий створений срільбою із зброої високого калібру, що збільшує віддачу і той же час робить контроль патерну більш важким.

Рисунок 4.10 – Патерни віддачі

Настутпним тестом буде взаємодія куль з гравцем. При попаданні кулі у гравця повинна знижуватись шкала життя. Тому для настутпного стенду було створено автоматичну турель, зображену на рисунку 4.11.

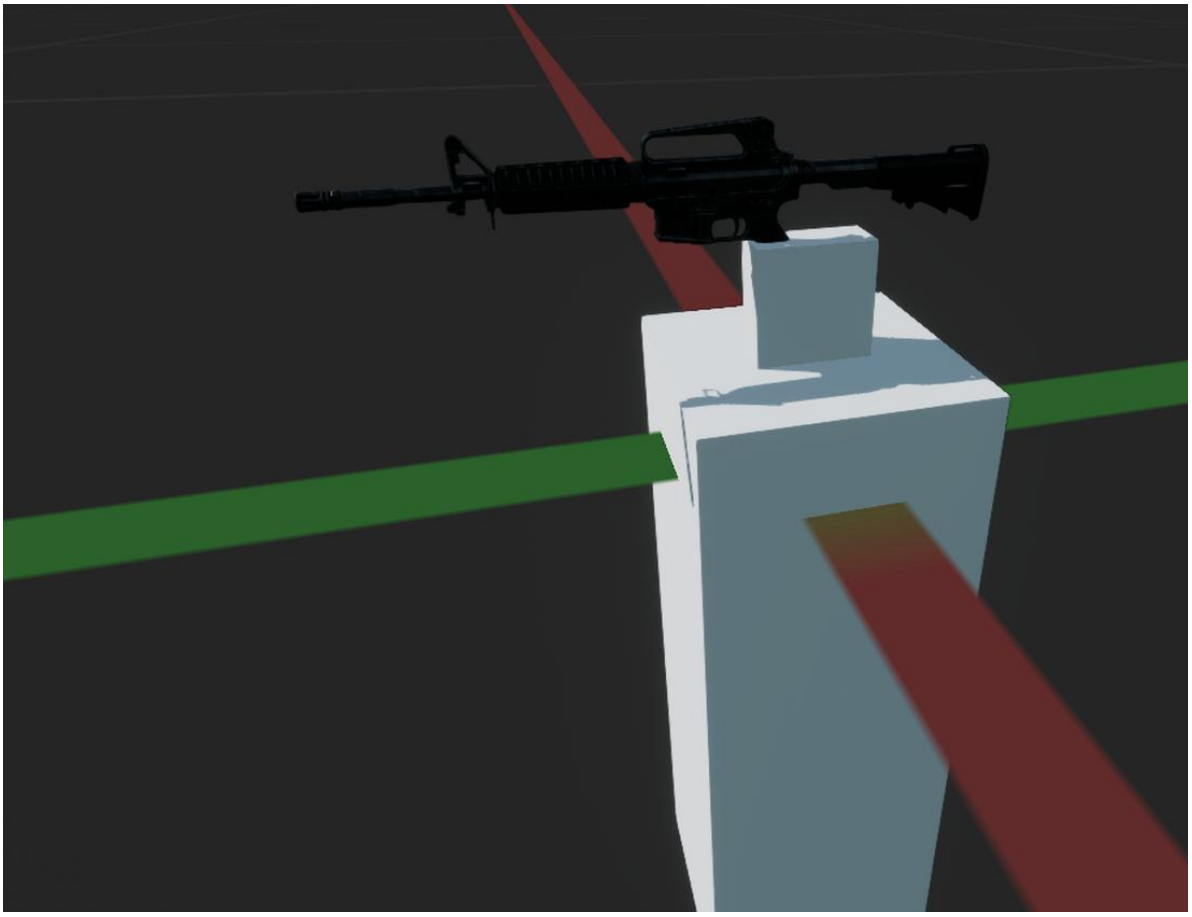

Рисунок 4.11 – Турель

Вона містить просту логіку: за таймером здійснювати посттріл на постійній основі. Логіку здійснення пострілу зображено на рисунку 4.12.

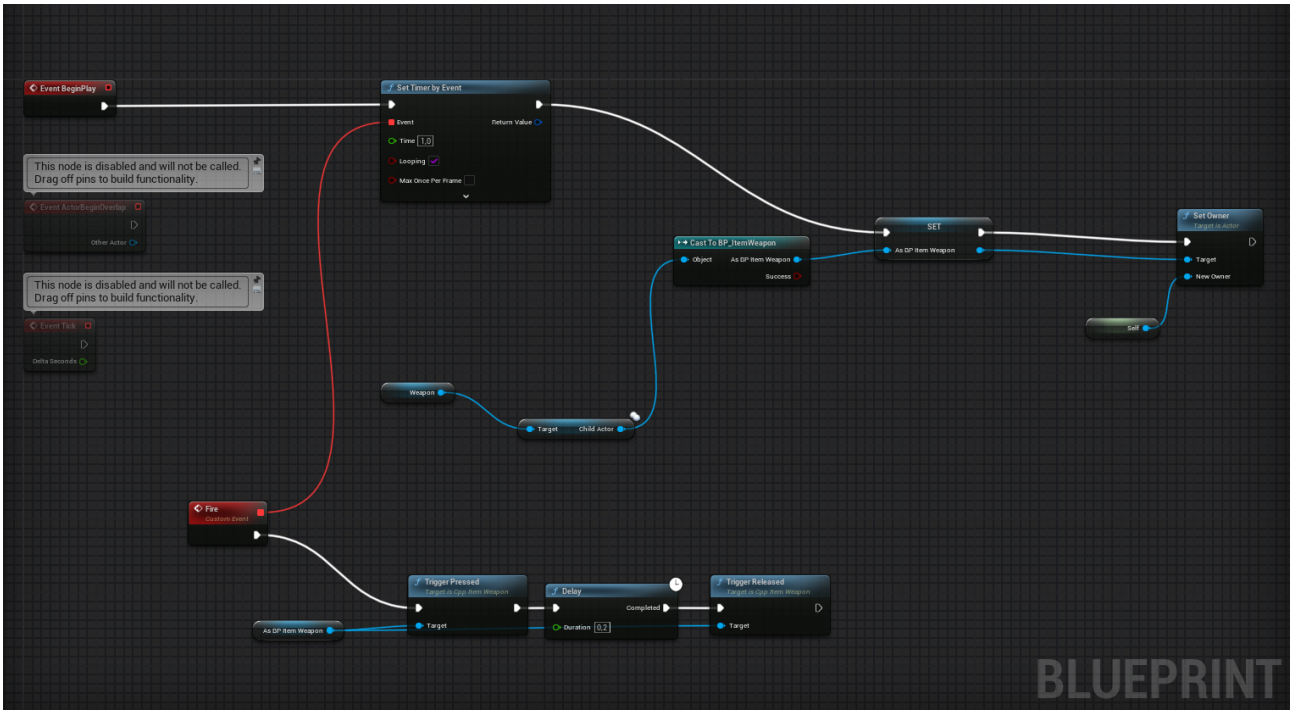

Рисунок 4.12 – Логіка турелі

Її буде розміщено у світі на певній відстані у напрямку гравця для перевірки функціоналу отримання урону.

При попаданні куля викликає подію на персонажі, що відповідає за зміну стану життя та шкали у інтерфейсі. Шкала життя зображена на рисунку 4.13.

Рисунок 4.12 – Шкала життя

Як видно з прикріпленої шкали, вона заповнена тільки частино, що свідчить про коректне отримання урону персонажем.

4.2 Розробка інструкції користувача

Інструкція користувача є важливим супровідним документом до програмного продукту, оскільки вона забезпечує кінцевому користувачеві можливість ефективно взаємодіяти з програмою без необхідності звертатися до технічної підтримки. Основне її призначення — надати зрозумілі пояснення щодо встановлення, налаштування та використання функціоналу програми. Вона допомагає уникнути помилок під час експлуатації, знижує ризик виникнення проблем, пов’язаних із несумісністю або неправильним використанням, а також підвищує рівень задоволеності користувачів. Інструкція містить опис інтерфейсу, підсказки для початку гри, пояснення призначення кнопок, полів або інструментів, а також способи реагування на типові помилки. Окрему увагу в інструкції приділяють технічним вимогам, які дають змогу оцінити, чи

відповідає пристрій користувача необхідним параметрам для коректної роботи програми. Це особливо актуально для ігрових продуктів, де продуктивність напряму залежить від конфігурації системи. Таким чином, інструкція є не лише засобом комунікації між розробником і користувачем, а й інструментом, що сприяє доступності, зручності та стабільній роботі програмного забезпечення.

В інструкцію користувача обов’язково входять вимоги до апаратного забезпечення, що є необхідними для стабільної роботи програмного продукту. Ці вимоги дозволяють користувачам визначити, чи їхня система здатна забезпечити належну продуктивність при запуску гри та уникнути можливих проблем з сумісністю або недостатньою потужністю для коректного виконання програми. У випадку цієї гри, мінімальні апаратні вимоги включають:

1. Відеокарта: NVIDIA GeForce GTX 1070.
2. Процесор: AMD Ryzen 5 3600.
3. Оперативна пам'ять: 16 ГБ DDR4 RAM,.

Рекомендовані системні вимоги для забезпечення оптимальної продуктивності та найкращого візуального досвіду під час гри включають:

1. Відеокарта: NVIDIA GeForce RTX 3060.
2. Процесор: AMD Ryzen 5 5600.
3. Оперативна пам'ять: 16 ГБ DDR4.

Рекомендованою відеокартою є відеокарта серії RTX. Її наявність дозволить використовувати функціонал трасування променів для збільшення реалістичності графіки.

Додатково рекомендується SSD, що зменшить час завантаження гри, рівнів, та покращить завантаження об’єктів у відкритому світі.

Після інсталяції гри через платформу Steam, користувач переходить до головного меню програмного забезпечення. На цьому етапі, перед початком гри, рекомендується налаштувати параметри графіки відповідно до конфігурації комп'ютера для досягнення оптимальної продуктивності. Для цього необхідно перейти в меню налаштувань графіки, де можна вибрати роздільну здатність екрану, рівень деталізації текстур, параметри освітлення та інші налаштування,

що впливають на продуктивність гри. Після того, як параметри графіки будуть налаштовані відповідно до можливостей системи, користувач може натиснуттти кнопку «Почати гру», щоб розпочати гру.

Усі необхідні підсказки та туторіали для користувачів передбачені безпосередньо в самій грі, щоб забезпечити плавне і зрозуміле навчання новачків і зручність для досвідчених гравців. Під час проходження гри гравці отримують покрокові інструкції, які допомагають ознайомитись із основними механіками та функціями гри, поступово вводячи їх у всі аспекти ігрового процесу. Це включає підказки щодо управління персонажем, використання інвентарю, взаємодії з об'єктами, а також особливості застосування різних ігрових механік, таких як стрільба, вибір зброї чи використання спеціальних навичок. Усі ці підказки подаються інтуїтивно зрозуміло і зручним для сприйняття способом, що дозволяє швидко освоїтися в грі.

Туторіали інтегровані в гру таким чином, щоб новачки могли легко освоїти базові навички без надмірної складності, а досвідчені гравці мали змогу швидко згадати або уточнити правила гри. Це дає можливість гравцям повернутися до навчальних матеріалів в будь-який момент, коли вони відчувають необхідність в додаткових поясненнях. Крім того, у будь-який момент можна звернутися до розділу «Довідка», де знайдуться більш детальні інструкції та пояснення щодо специфічних функцій гри, що дозволяють отримати необхідну інформацію для ефективної гри та уникнути непорозумінь.

4.3 Висновки

У четвертому розділі було проведено тестування програмних модулів, які реалізують механіку стрільби в шутер-грі. Перевірено реакцію програмного забезпечення на різні ситуації, такі як взаємодія з об'єктами в середовищі, зміну траєкторії польоту кулі при проходженні через різні перешкоди, а також вплив фізичних властивостей на точність і ефективність стрільби. Особливу увагу було приділено тестуванню рикошету куль при попаданні в різні матеріали з різними характеристиками.

Крім того, було описано інструкцію користувача, яка детально описує технічні вимоги для запуску гри, налаштування параметрів графіки та інтерфейсу. Інструкція також містить покрокову інформацію щодо процесу інсталяції гри через Steam, а також допомагає користувачам налаштувати гру відповідно до конфігурації їхніх комп'ютерів.

\

ВИСНОВКИ

У рамках виконання бакалаврської кваліфікаційної роботи була розроблена шутер-гра "Rock In A Hard Place: Lonesome Road", основною метою якої стало створення програмного забезпечення з високим рівнем реалістичності та інтерактивності. Проєкт реалізовано з використанням сучасного ігрового рушія Unreal Engine 5 та мови програмування C++, що дозволило забезпечити високу продуктивність, деталізацію графіки та зручність інтеграції складних ігрових механік.

У ході роботи був розроблений алгоритм симуляції балістики куль, що враховує такі фізичні параметри, як гравітація, опір повітря та швидкість кулі, що дозволяє досягти максимальної реалістичності стрільби. Окрім цього, реалізовано модулі управління персонажем, які включають обробку рухів, стрибків та взаємодії з об’єктами, а також модулі управління зброєю, що відповідають за стрільбу, перезарядку, зміну озброєння та облік фізичних властивостей. Такі модулі створили динамічний і захопливий ігровий процес, що відповідає сучасним вимогам індустрії інтерактивних розваг.

Особливу увагу було приділено розробці графічного інтерфейсу, який створено в мінімалістичному стилі для забезпечення зручності використання та інтуїтивності. Адаптивний дизайн інтерфейсу дозволяє гравцю зручно налаштовувати параметри гри, відстежувати стан персонажа та взаємодіяти з елементами оточення без відволікання від основного ігрового процесу. Інтерфейс розроблено із застосуванням візуального інструменту Blueprints, що забезпечило його ефективну інтеграцію в ігрову механіку.

Тестування програмного продукту здійснювалося у кілька етапів: було перевірено функціональність основних модулів, коректність алгоритмів симуляції балістики, а також інтерактивність графічного інтерфейсу. Проведений аналіз показав, що програмне забезпечення повністю відповідає поставленим вимогам технічного завдання та забезпечує стабільну роботу навіть за інтенсивних ігрових сценаріїв.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Пахолюк В.Б., Катєльніков Д.І. Пристрої з вбудованим штучним інтелектом. Матеріали IV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів і студентів "Комп`ютерні ігри та мультимедіа як інноваційний підхід до комунікації -2024", Одеса, 26-27 вересня 2024 р. - Одеса, Видавництво ОНТУ, 2024 р. С. 341-342
2. Лисенко Г.Л. Методичні вказівки до оформлення курсових проектів (робіт) у Вінницькому національному технічному університеті /Уклад. Г.Л. Лисенко, А.Г. Буда, Р.Р. Обертюх, – Вінниця: ВНТУ, 2006. – 60 с.
3. Unreal Engine 5 Documentation. [Електронний ресурс] – Режим доступу: https://docs.unrealengine.com (дата звернення: 02.05.2025).
4. Unreal Engine GitHub Repository. [Електронний ресурс] – Режим доступу: https://github.com/EpicGames/UnrealEngine (дата звернення: 02.05.2025).
5. C++ Programming Language Documentation. [Електронний ресурс] – Режим доступу: https://cplusplus.com (дата звернення: 02.05.2025).
6. Nanite in Unreal Engine. [Електронний ресурс] – Режим доступу: https://www.unrealengine.com/en-US/tech-blog (дата звернення: 02.05.2025).
7. Lumen in Unreal Engine. [Електронний ресурс] – Режим доступу: https://www.unrealengine.com/en-US/tech-blog (дата звернення: 02.05.2025).
8. Godot Engine Documentation. [Електронний ресурс] – Режим доступу: https://docs.godotengine.org (дата звернення: 02.05.2025).
9. Unity Engine. [Електронний ресурс] – Режим доступу: https://unity.com/ (дата звернення: 02.05.2025).
10. Blender 3D Modeling Software Documentation. [Електронний ресурс] – Режим доступу: https://docs.blender.org/manual/en/latest/ (дата звернення: 02.05.2025).

11. Vulkan Graphics API Documentation. [Електронний ресурс] – Режим доступу: https://vulkan.lunarg.com/doc/sdk/latest/windows/ (дата звернення: 02.05.2025).
12. OpenGL Graphics API Documentation. [Електронний ресурс] – Режим доступу: https://www.opengl.org/documentation/ (дата звернення: 02.05.2025).
13. Unity Engine Documentation. [Електронний ресурс] – Режим доступу: https://docs.unity3d.com (дата звернення: 02.05.2025).
14. Unity Engine Documentation. [Електронний ресурс] – Режим доступу: https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/Engine/GameFramework/UProjectileMovementComponent (дата звернення: 02.05.2025).
15. Unity Engine Documentation. [Електронний ресурс] – Режим доступу: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-5-documentation (дата звернення: 02.05.2025).
16. Unreal Engine Shooter game basics. [Електронний ресурс] – Режим доступу: https://dev.epicgames.com/community/learning/tutorials/YBr/creating-your-first-person-shooter-game-in-unreal-engine-5 (дата звернення: 02.05.2025).
17. Visual Studio IDE Documentation. [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/visualstudio/ (дата звернення: 02.05.2025).
18. GitHub Guide for Developers. [Електронний ресурс] – Режим доступу: https://guides.github.com (дата звернення: 02.05.2025).
19. Microsoft Azure Cloud Services Overview. [Електронний ресурс] – Режим доступу: https://azure.microsoft.com/en-us/overview/ (дата звернення: 02.05.2025).
20. Steam Platform Overview. [Електронний ресурс] – Режим доступу: https://store.steampowered.com (дата звернення: 02.05.2025).

ДОДАТКИ

58

Додаток А – Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедрою ПЗ
д.т.н., проф. О. Н. Романюк
_____
" 25 " березня 2025 року

Технічне завдання
на бакалаврську кваліфікаційну роботу «розробка TRUE FIRST PERSON SHOOTER гри "Rock In A Hard Place: Lonesome Road" з використанням платформи Unreal Engine 5» за спеціальністю
121 – Інженерія програмного забезпечення

Керівник бакалаврської кваліфікаційної роботи:
_____ к.т.н., доцент Д. І. Катєльніков
"24" березня 2025 року
Виконав:
_____ студент гр. 1ПІ-21б В.Б. Пахолюк
"24" березня 2025 року

Вінниця – 2025 року

1. Найменування та галузь застосування

Бакалаврська кваліфікаційна робота: «розробка TRUE FIRST PERSON SHOOTER гри "Rock In A Hard Place: Lonesome Road" з використанням платформи Unreal Engine 5».

Галузь застосування – розваги.

2. Підстава для розробки.

Підставою для виконання бакалаврської кваліфікаційної’ роботи (БКР) є індивідуальне завдання на БКР та наказ №97 від «20» березня 2025 р. ректора по ВНТУ про закріплення тем БКР.

3. Мета та призначення розробки.

Метою дослідження є покращення взаємодії гравця з озброєнням, включаючи точність відтворення балістичних характеристик і фізичної взаємодії об’єктів у тривимірному просторі за рахунок системи True First Person.ідвищення ефективності вивчення користувачем польської мови шляхом розробки та використання спеціалізованої мобільної навчальної системи з теоретичним матеріалом і тестами, що дозволить підвищити рівень володіння новою мовою.

Призначення роботи – розробка та програмна реалізація відеогри.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БКР.

1. Sonic Workflow. Unreal Engine 5 Game Programming Design Patterns in C++, Java, C#, and Blueprints, 2021.

2. Stephen Ulibarri. Unreal Engine 5 Collision Essentials: Understanding Unreal Engine’s Collision Framework, 2024.

3. Devin Sherry. Elevating Game Experiences with Unreal Engine 5: Bring your game ideas to life using the new Unreal Engine 5 and C++, 2022.

4. Войтко В.В. Навчальна система для вивчення і тестування з польської мови / В. В. Войтко, Г. Б. Ракитянська, С. В. Бевз, С. М. Бурбело, А. М. Манич // Матеріали Всеукраїнської науково-практичної інтернет-конференції "Молодь в науці: дослідження, проблеми, перспективи - 2022", Секція - Інформаційні технології та комп‘ютерна інженерія. [Електронний ресурс] – Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2022/paper/viewFile/16201/13643.

5. Технічні вимоги

Середовище розробки – Visual Studio, Unreal engine; мова програмування – C++.

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред’являється по закінченню робіт:

1. Пояснювальна записка до БКР;
2. Технічне завдання;
3. Лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз завдання і вибір методу вирішення поставленої задачі дослідження	26.03.2025 - 29.03.2025
2	Розробка дизайну графічного інтерфейсу	10.04.2025 - 19.04.2025
3	Вибір середовища та мови розробки	20.04.2025 - 24.04.2025
5	Розробка модулів програми	27.04.2025 - 11.05.2025
5	Тестування програми	11.05.2025 - 15.05.2025
6	Оформлення матеріалів до захисту БКР	16.05.2025 - 30.05.2025

10. Порядок контролю та прийняття

Виконання етапів бакалаврської кваліфікаційна роботи контролюється керівником згідно з графіком виконання роботи.

Прийняття бакалаврської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

62

Додаток Б – Перевірка на плагіат

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: розробка TRUE FIRST PERSON SHOOTER гри "Rock In A Hard Place: Lonesome'

Тип роботи: _____ бакалаврська кваліфікаційна робота _____
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ: кафедра програмного забезпечення, ФІТКІ
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 5 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

■ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____ _____
(прізвище, ініціали, посада) (підпис)

_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Черноволик Г. О.

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Катєльніков Д. І. к.т.н., доц. каф ПЗ
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Пахолюк В. Б.
(підпис) (прізвище, ініціали)

Додаток В – Лістинг програми

```
// RockInAHardPlace



#include "cpp_Projectile.h"
#include "GameFramework/ProjectileMovementComponent.h"
#include "Components/BoxComponent.h"
#include "DrawDebugHelpers.h"
#include "Engine/World.h"
#include "Components/StaticMeshComponent.h"
#include <Kismet/KismetMathLibrary.h>
#include <Kismet/GameplayStatics.h>


// Sets default values
Acpp_Projectile::Acpp_Projectile()
{
	// Set this actor to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
	PrimaryActorTick.bCanEverTick = false;


	CollisionBox = CreateDefaultSubobject<UBoxComponent>(TEXT("CollisionBox"));
	SetRootComponent(CollisionBox);
	CollisionBox->SetCollisionObjectType(ECollisionChannel::ECC_WorldDynamic);
	CollisionBox->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
	CollisionBox->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
	CollisionBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_Visibility, ECollisionResponse::ECR_Overlap);
	CollisionBox->SetCollisionResponseToChannel(ECollisionChannel::ECC_WorldStatic, ECollisionResponse::ECR_Overlap);
```

```
	Mesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Mesh"));

	Mesh->SetupAttachment(RootComponent);


	ProjectileMovementComponent =
CreateDefaultSubobject<UProjectileMovementComponent>(TEXT("ProjectileMovementCompone
nt"));

	ProjectileMovementComponent->bRotationFollowsVelocity = true;

	ProjectileMovementComponent->bShouldBounce = false;




}


// Called when the game starts or when spawned

void Acpp_Projectile::BeginPlay()

{

	Super::BeginPlay();

	GetWorld()->GetTimerManager().SetTimer(LifeTimerHandle, this,
&ThisClass::DestroyProjectile, TimeToLive, false);

	CollisionBox->MoveIgnoreActors.Add(Owner);

	CollisionBox->OnComponentBeginOverlap.AddDynamic(this,
&ThisClass::OnBoxComponentBeginOverlap);

	CollisionBox->OnComponentEndOverlap.AddDynamic(this,
&ThisClass::OnBoxComponentEndOverlap);

}


bool Acpp_Projectile::ComputeExitLocation(const FHitResult& ImpactResult, const FVector&
ImpactVelocity)

{

	FVector ImpactNormalized = ImpactVelocity.GetSafeNormal() * 500 +
ImpactResult.Location;

	FVector ResultVector;
```

```
        AActor* HitActor = ImpactResult.GetActor();


        //Trace
// Trace parameters
FVector Start = ImpactNormalized; // Starting point of the trace
FVector End = ImpactResult.ImpactPoint; // Ending point of the trace
float Radius = 0.5f;                // Sphere radius
FCollisionShape SphereShape = FCollisionShape::MakeSphere(Radius);

// Object types to query
FCollisionObjectQueryParams ObjectQueryParams;
ObjectQueryParams.AddObjectTypesToQuery(ECC_WorldStatic);   // Static objects
ObjectQueryParams.AddObjectTypesToQuery(ECC_WorldDynamic); // Dynamic objects
ObjectQueryParams.AddObjectTypesToQuery(ECC_PhysicsBody);  // Physics-enabled objects

// Trace parameters
FCollisionQueryParams TraceParams(SCENE_QUERY_STAT(MultiSphereTrace), false, this);
TraceParams.AddIgnoredActor(this); // Ignore self to avoid false positives

// To store the results
TArray<FHitResult> HitResults;

// Perform the sweep
        bool bHit = GetWorld()->SweepMultiByObjectType(
                HitResults,
                Start,
                End,
                FQuat::Identity,
```

```
            ObjectQueryParams,
            SphereShape,
            TraceParams
    );


    for (const FHitResult& Hit : HitResults)
    {
        AActor* SecondHitActor = Hit.GetActor();
        if (SecondHitActor == HitActor)
        {
            ResultVector = Hit.Location;
            if (ComputePenetration(End, ResultVector)<=PenetrationMax) return true;
        }
    }

    ResultVector = FVector();
    return false;
}

bool Acpp_Projectile::Ricochet(const FHitResult& ImpactResult, const FVector& ImpactVelocity)
{
    const float RicochetThreshold = -0.3f; // Example threshold; adjust as needed
    FVector ImpactNormal = ImpactResult.ImpactNormal; // Assuming ImpactNormal is part of FHitResult
    FVector Velocity = ProjectileMovementComponent->Velocity; // Get current velocity

    float MaxSpeedLoss = 0.4f;
    float MinSpeedLoss = 1.0f;
```

```
	float IncidenceAngle = FVector::DotProduct(ImpactVelocity.GetSafeNormal(), ImpactNormal);

	float VelocityLossFactor = UKismetMathLibrary::MapRangeClamped(IncidenceAngle, RicochetThreshold, 0.0f, MaxSpeedLoss, MinSpeedLoss);



	// Calculate the reflection vector

	FVector ReflectionVector = ImpactVelocity - 2 * FVector::DotProduct(ImpactVelocity, ImpactNormal) * ImpactNormal;



	// Scale the reflection vector by the VelocityLossFactor

	ReflectionVector = ReflectionVector.GetSafeNormal() * ImpactVelocity.Size() * VelocityLossFactor;



	// Update the projectile's velocity

	ProjectileMovementComponent->Velocity = ReflectionVector;



	// Check if the angle is shallow enough for a ricochet

	bool bShouldRicochet = (IncidenceAngle > RicochetThreshold);

	return bShouldRicochet;

}



float Acpp_Projectile::ComputePenetration(FVector& Entry, FVector& Exit)

{

	float PenetrationLenght = (Entry - Exit).Length();

	ProjectileMovementComponent->Velocity *= UKismetMathLibrary::MapRangeClamped(PenetrationLenght, 0.0f, PenetrationMax, 1.0f, 0.4f);

	return PenetrationLenght;

}
```

```
void Acpp_Projectile::OnBoxComponentBeginOverlap(UPrimitiveComponent*
OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)

{

	FVector ImpactLocation = SweepResult.ImpactPoint;

	SpawnDecal(SweepResult);

	if (ComputeExitLocation(SweepResult, ProjectileMovementComponent->Velocity))
DealDamage(ImpactLocation, OtherActor, true);

	else if (Ricochet(SweepResult, ProjectileMovementComponent->Velocity))
DealDamage(ImpactLocation, OtherActor, true);

	else DealDamage(ImpactLocation, OtherActor, false);

}


void Acpp_Projectile::OnBoxComponentEndOverlap(UPrimitiveComponent*
OverlappedComponent, AActor* OtherActor, UPrimitiveComponent* OtherComp, int32
OtherBodyIndex)

{

}


void Acpp_Projectile::DealDamage(FVector& ImpactLocation, AActor* OtherActor, bool
bShouldStayAlive)

{

	// Apply damage to the impacted actor

	if (OtherActor && OtherActor != this) // Avoid self-damage

	{

		UGameplayStatics::ApplyDamage(

			OtherActor,

			50.0f,		// Damage value

			GetInstigatorController(),

			this,

			nullptr		// Damage type class (can specify custom damage types)

		);
```

```
        }

        // Optionally destroy the projectile
        if (!bShouldStayAlive)
        {
                DestroyProjectile();
        }
}

void Acpp_Projectile::SpawnDecal(const FHitResult& ImpactResult)
{
        if (DecalMaterial) // Ensure the DecalMaterial is valid
        {
                FVector DecalSize = FVector(10.0f, 10.0f, 10.0f); // Adjust size as needed

                // Calculate the rotation based on the impact normal
                FRotator DecalRotation = ImpactResult.ImpactNormal.Rotation()*-1;

                // Spawn the decal at the impact location
                UGameplayStatics::SpawnDecalAtLocation(
                        GetWorld(),
                        DecalMaterial,
                        DecalSize,
                        ImpactResult.ImpactPoint,
                        DecalRotation,
                        10.0f
                );
        }
}
```

```
void Acpp_Projectile::DestroyProjectile()
{
	Destroy();
}

// Called every frame
void Acpp_Projectile::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);

}

void Acpp_Projectile::SetSpeed(float inMaxSpeed, float inInitialSpeed)
{
	ProjectileMovementComponent->InitialSpeed = inInitialSpeed;
	ProjectileMovementComponent->MaxSpeed = inMaxSpeed;
	ProjectileMovementComponent->SetVelocityInLocalSpace(FVector(inInitialSpeed, 0, 0));
}

// RockInAHardPlace

#include "cpp_EquipmentComponent.h"
#include "RockInAHardPlace/Character/cppRIHPCharacter.h"
#include "RockInAHardPlace/Enums/cpp_SlotEnums.h"
#include "RockInAHardPlace/Items/Equippable/Weapons/cpp_ItemWeapon.h"
#include "RockInAHardPlace/Enums/cpp_EWeaponFiremodes.h"
#include "Curves/CurveFloat.h"
#include "Kismet/KismetMathLibrary.h"
```

```
#include "Camera/CameraComponent.h"

#include "RockInAHardPlace/Character/cpp_Character_AnimInstance.h"

#include "RockInAHardPlace/cpp_Mag.h"

#include "Components/SkinnedMeshComponent.h"


// Sets default values for this component's properties

Ucpp_EquipmentComponent::Ucpp_EquipmentComponent()

{

	// Set this component to be initialized when the game starts, and to be ticked every frame. You can turn these features

	// off to improve performance if you don't need them.

	PrimaryComponentTick.bCanEverTick = true;



	RecoilVerticalInterpFun.BindUFunction(this, FName("RecoilProgressVertical"));

	RecoilHorizontalInterpFun.BindUFunction(this, FName("RecoilProgressHorizontal"));

	RecoilTimeline = CreateDefaultSubobject<UTimelineComponent>(TEXT("RecoilTimeline"));

}


void Ucpp_EquipmentComponent::Initialization(AcppRIHPCharacter* character)

{

	CharacterRef = character;

}


void Ucpp_EquipmentComponent::GrabMag(int inWhichMag)

{

	FAttachmentTransformRules AttachRules (EAttachmentRule::KeepWorld, EAttachmentRule::KeepWorld, EAttachmentRule::KeepWorld, true);

	//CurrentWeapon->MagazineRef->AttachToComponent(CharacterRef->GetMesh(), AttachRules, FName("GrabBone_lSocket"));
```

```
	CurrentWeapon->MagazineRef->AttachToComponent(CurrentWeapon->GetSkeletalMesh(), AttachRules, FName("S_Mag"));

}


void Ucpp_EquipmentComponent::DropMag(int inWhichMag)

{

	FDetachmentTransformRules DeattachRules(EDetachmentRule::KeepWorld, EDetachmentRule::KeepWorld, EDetachmentRule::KeepWorld, false);

	CurrentWeapon->MagazineRef->DetachFromActor(DeattachRules);

	CurrentWeapon->MagazineRef->EnablePhysics(true);

}


void Ucpp_EquipmentComponent::SpawnMag()

{

	if (NewMag) NewMag->Destroy();

	FTransform SpawnTransform = CharacterRef->GetActorTransform();

	NewMag = GetWorld()->SpawnActor<Acpp_Mag>(CurrentWeapon->MagazineClass, SpawnTransform);

	FAttachmentTransformRules AttachRules(EAttachmentRule::SnapToTarget, EAttachmentRule::SnapToTarget, EAttachmentRule::KeepWorld, true);

	NewMag->AttachToComponent(CharacterRef->GetMesh(), AttachRules, FName("GrabBone_lSocket"));

}


void Ucpp_EquipmentComponent::InsertMag()

{

	FAttachmentTransformRules AttachRules(EAttachmentRule::KeepWorld, EAttachmentRule::KeepWorld, EAttachmentRule::KeepWorld, true);

	NewMag->AttachToComponent(CurrentWeapon->GetSkeletalMesh(), AttachRules);

	Acpp_Mag* tempMag = CurrentWeapon->MagazineRef;

	CurrentWeapon->MagazineRef = NewMag;
```

```
	NewMag = tempMag;

}


void Ucpp_EquipmentComponent::DeleteMag()

{

	NewMag->Destroy();

}


void Ucpp_EquipmentComponent::ReloadFinish()

{

	FAttachmentTransformRules AttachRules(EAttachmentRule::KeepWorld, EAttachmentRule::KeepWorld, EAttachmentRule::KeepWorld, true);

	CurrentWeapon->MagazineRef->AttachToComponent(CurrentWeapon->GetSkeletalMesh(), AttachRules);

}




// Called when the game starts

void Ucpp_EquipmentComponent::BeginPlay()

{

	FTransform SpawnTransform = CharacterRef->GetActorTransform();

	FAttachmentTransformRules AttachRules = FAttachmentTransformRules(EAttachmentRule::SnapToTarget, EAttachmentRule::SnapToTarget, EAttachmentRule::KeepWorld, true);

	if (PrimaryWeaponClass)

	{

		PrimaryWeaponSlot = GetWorld()->SpawnActor<Acpp_ItemWeapon>(PrimaryWeaponClass, SpawnTransform);

		PrimaryWeaponSlot->AttachToComponent(CharacterRef->GetMesh(), AttachRules, PrimarySocketName);

	}
```

```
	if (SecondaryWeaponClass)

	{

		SecondaryWeaponSlot = GetWorld()-
>SpawnActor<Acpp_ItemWeapon>(SecondaryWeaponClass, SpawnTransform);

		SecondaryWeaponSlot->AttachToComponent(CharacterRef->GetMesh(),
AttachRules, SecondarySocketName);

	}

	if (SidearmWeaponClass)

	{

		SidearmWeaponSlot = GetWorld()-
>SpawnActor<Acpp_ItemWeapon>(SidearmWeaponClass, SpawnTransform);

		SidearmWeaponSlot->AttachToComponent(CharacterRef->GetMesh(), AttachRules,
SidearmSocketName);

	}


	//Add Slots to map

	WeaponSlotsMap.Add(EWeaponSlotsEnum::PrimarySlot, PrimaryWeaponSlot);

	WeaponSlotsMap.Add(EWeaponSlotsEnum::SecondarySlot, SecondaryWeaponSlot);

	WeaponSlotsMap.Add(EWeaponSlotsEnum::SidearmSlot, SidearmWeaponSlot);

	//Add Sockets to map

	WeaponSocketsMap.Add(EWeaponSlotsEnum::PrimarySlot, PrimarySocketName);

	WeaponSocketsMap.Add(EWeaponSlotsEnum::SecondarySlot, SecondarySocketName);

	WeaponSocketsMap.Add(EWeaponSlotsEnum::SidearmSlot, SidearmSocketName);

}



void Ucpp_EquipmentComponent::ConvertTransformValuesToUE(const FRotator& inRotator,
const FVector& inLocation, FRotator& outRotator, FVector& outLocation)

{

	outRotator = FRotator(inRotator.Pitch, (180.0 - inRotator.Yaw),-inRotator.Roll);

	outLocation = FVector(inLocation.X, -inLocation.Y, inLocation.Z);
```

```
}

void Ucpp_EquipmentComponent::PlayAnimMonCharacter(UAnimMontage* inAnimToPlay)
{
	CharacterRef->GetMesh()->GetAnimInstance()->Montage_Play(inAnimToPlay);
}

void Ucpp_EquipmentComponent::RecoilProgressVertical(float inRecoil)
{
	// Recoil should be negative to face the right direction
	if (!RecoilTimeline->IsReversing()&&bIsShooting)
	{
		float DeltaTime = GetWorld()->GetDeltaSeconds()*100;
		float Recoil = inRecoil * DeltaTime;
		RecoilToCompensate = RecoilToCompensate + Recoil;
		CharacterRef->AddControllerPitchInput(-Recoil);
		if (!bIsAimingDownsights)
		{
			OffsetRecoilToCompensate = OffsetRecoilToCompensate + Recoil;
			OffsetRecoilToCompensate = FMath::Clamp(OffsetRecoilToCompensate, -
15.0f, 15.0f);
			FVector2D WeaponOffsetRecoil{};
			WeaponOffsetRecoil.Y += -Recoil;
			OffsetWeapon(WeaponOffsetRecoil);
		}
	}
}

void Ucpp_EquipmentComponent::RecoilProgressHorizontal(float inRecoil)
```

```
{
	if (!RecoilTimeline->IsReversing() && bIsShooting)
	{
		float DeltaTime = GetWorld()->GetDeltaSeconds() * 100;
		float Recoil = inRecoil * DeltaTime;
		RecoilToCompensateH = RecoilToCompensateH -inRecoil * DeltaTime;
		CharacterRef->AddControllerYawInput(inRecoil * DeltaTime);
		if (!bIsAimingDownsights)
		{
			OffsetRecoilToCompensateH = OffsetRecoilToCompensateH + Recoil;
			OffsetRecoilToCompensateH = FMath::Clamp(OffsetRecoilToCompensateH,
-15.0f, 15.0f);
			FVector2D WeaponOffsetRecoil{};
			WeaponOffsetRecoil.X += inRecoil * DeltaTime;
			OffsetWeapon(WeaponOffsetRecoil);
		}
	}
}

void Ucpp_EquipmentComponent::BindRecoilCurve(Acpp_ItemWeapon* inWeapon)
{
	RecoilTimeline->Stop();
	RecoilTimeline->Deactivate();
	RecoilTimeline->DestroyComponent();
	RecoilTimeline = NewObject<UTimelineComponent>(this,
UTimelineComponent::StaticClass());
	RecoilTimeline->RegisterComponent();
	RecoilTimeline->AddInterpFloat(inWeapon->RecoilCurveVertical,
RecoilVerticalInterpFun, FName("Alpha"));
```

```
	RecoilTimeline->AddInterpFloat(inWeapon->RecoilCurveHorizontal, RecoilHorizontalInterpFun, FName("Alphab"));

}


void Ucpp_EquipmentComponent::CompensateOffsetRecoil()

{

	FVector2D RecoilCompensation{};

	float DeltaTime = GetWorld()->GetDeltaSeconds();

	RecoilCompensation.Y+= OffsetRecoilToCompensate * DeltaTime * RecoilReverseTime;

	OffsetRecoilToCompensate -= OffsetRecoilToCompensate * DeltaTime * RecoilReverseTime;

	RecoilCompensation.X -= OffsetRecoilToCompensateH * DeltaTime * RecoilReverseTime;

	OffsetRecoilToCompensateH -= OffsetRecoilToCompensateH * DeltaTime * RecoilReverseTime;

	if (!bIsAimingDownsights) {

		OffsetWeapon(RecoilCompensation);

	}

}


// Called every frame

void Ucpp_EquipmentComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)

{

	Super::TickComponent(DeltaTime, TickType, ThisTickFunction);



	// ...

	if (!bIsShooting && (HipfireOffset.X != 0 || HipfireOffset.Y != 0))

	{

		HipfireOffset = FMath::Vector2DInterpTo(HipfireOffset, FVector2D(0.0f), GetWorld()->GetDeltaSeconds(), 5.0f);
```

```
		HipfireAlpha = FMath::FInterpTo(HipfireAlpha, 0.0f, GetWorld()->GetDeltaSeconds(), 10.0f);

	}

	else if (bIsShooting)

	{

		//Hipfire recoil

		HipfireOffset = FMath::Vector2DInterpTo(HipfireOffset, HipfireOffsetModified, GetWorld()->GetDeltaSeconds(), 10.0f);

		HipfireAlpha = FMath::FInterpTo(HipfireAlpha, 1.0f, GetWorld()->GetDeltaSeconds(), 10.0f);

	}

	if (bResetRecoil)

	{

		if (RecoilToCompensate > 0)

		{

			CharacterRef->AddControllerPitchInput(RecoilToCompensate * DeltaTime * RecoilReverseTime);

			RecoilToCompensate = RecoilToCompensate - RecoilToCompensate * DeltaTime * RecoilReverseTime;

		}

		if (RecoilToCompensateH != 0)

		{

			CharacterRef->AddControllerYawInput(RecoilToCompensateH * DeltaTime * RecoilReverseTime);

			RecoilToCompensateH = RecoilToCompensateH - RecoilToCompensateH * DeltaTime * RecoilReverseTime;

		}

		if (RecoilToCompensate <= 0.01&&((RecoilToCompensateH>=-0.01)&&(RecoilToCompensateH <= 0.01)))

		{

			bResetRecoil = false;

		}
```

```
		else
		{
			CompensateOffsetRecoil();
		}
	}
	if (bShouldChangeFOV)
	{
		ChangeFOV(DeltaTime);
	}
	if (bIsAimingDownsights)
	{
		WeaponAimOffset.Y = (FMath::FInterpTo(WeaponAimOffset.Y, 0, DeltaTime, CurrentWeapon->GetAimdownsightsSpeed()));
		WeaponAimOffset.X = (FMath::FInterpTo(WeaponAimOffset.X, 0, DeltaTime, CurrentWeapon->GetAimdownsightsSpeed()));
		OffsetRecoilToCompensate = (FMath::FInterpTo(OffsetRecoilToCompensate, 0, DeltaTime, CurrentWeapon->GetAimdownsightsSpeed()));
		OffsetRecoilToCompensateH = (FMath::FInterpTo(OffsetRecoilToCompensateH, 0, DeltaTime, CurrentWeapon->GetAimdownsightsSpeed()));
	}
}

void Ucpp_EquipmentComponent::ChangeFOV(float DeltaTime)
{
	if (CurrentWeapon)
	{
		float AimSpeed = (bIsChangingSights&&FMath::IsNearlyEqual(AimDownsightsWeight, 1, 0.001f)) ? (1 / ChangeSightTime):CurrentWeapon->GetAimdownsightsSpeed();

		UCameraComponent* Camera = CharacterRef->GetFirstPersonCamera();
```

```
            float CurrentFOV = Camera->FieldOfView;

            float SightZoom = CurrentWeapon->GetSightMagnification();

            float DesiredFOV = bIsAimingDownsights ? (CharacterRef->GetBaseFOV() /
SightZoom) : CharacterRef->GetBaseFOV();

            if (bIsAimingDownsights)

            {

                  Camera->SetFieldOfView(FMath::FInterpTo(CurrentFOV, DesiredFOV,
DeltaTime, AimSpeed));

                  AimDownsightsWeight = (FMath::FInterpTo(AimDownsightsWeight, 1,
DeltaTime, AimSpeed));

                  CharacterRef->sensitivity = (FMath::FInterpTo(CharacterRef->sensitivity,
1.0f / SightZoom, DeltaTime, AimSpeed));

            }

            else

            {

                  Camera->SetFieldOfView(FMath::FInterpTo(CurrentFOV, CharacterRef-
>GetBaseFOV(), DeltaTime, AimSpeed));

                  AimDownsightsWeight = (FMath::FInterpTo(AimDownsightsWeight, 0,
DeltaTime, AimSpeed));

                  CharacterRef->sensitivity = (FMath::FInterpTo(CharacterRef->sensitivity,
1.0f, DeltaTime, AimSpeed));

            }

            if (FMath::IsNearlyEqual(CurrentFOV, DesiredFOV, 0.1f)) {

                  bShouldChangeFOV = false;

                  bIsChangingSights = false;

            }

      }

}


void Ucpp_EquipmentComponent::OffsetWeapon(const FVector2D& inLook)

{

      if (!bIsAimingDownsights)
```

```
	{
		WeaponAimOffset.Y += inLook.Y;
		WeaponAimOffset.X += inLook.X;
		WeaponAimOffset = WeaponAimOffset.ClampAxes(-15.0f, 15.0f);
	}
}

void Ucpp_EquipmentComponent::HolsterWeapon(bool bSwitchWeapon)
{
	if (CurrentWeaponSlot != EWeaponSlotsEnum::NullSlot)
	{
		Acpp_ItemWeapon* WeaponSlot = *WeaponSlotsMap.Find(CurrentWeaponSlot);
		FAttachmentTransformRules AttachRules = FAttachmentTransformRules(EAttachmentRule::SnapToTarget, EAttachmentRule::SnapToTarget, EAttachmentRule::KeepWorld, true);
		WeaponSlot->AttachToComponent(CharacterRef->GetMesh(), AttachRules, *WeaponSocketsMap.Find(CurrentWeaponSlot));
		CurrentWeaponSlot = EWeaponSlotsEnum::NullSlot;
		bIsWeaponDrawn = bSwitchWeapon;
	}
}

void Ucpp_EquipmentComponent::PulloutWeapon(EWeaponSlotsEnum Slot)
{
	FireButtonReleased();
	ReleaseAimDownsights();
	Acpp_ItemWeapon* Weapon = *WeaponSlotsMap.Find(Slot);
	if (CurrentWeaponSlot != Slot)
	{
		bIsWeaponDrawn = true;
```

```
		FAttachmentTransformRules AttachRules = FAttachmentTransformRules(EAttachmentRule::SnapToTarget, EAttachmentRule::SnapToTarget, EAttachmentRule::KeepWorld, true);

		Weapon->AttachToComponent(CharacterRef->GetMesh(), AttachRules, HandSocket);

		FVector ConvertedLocation=Weapon->AttachmentOffsetLocation;

		FRotator ConvertedRotation=Weapon->AttachmentOffsetRotation;

		ConvertTransformValuesToUE(Weapon->AttachmentOffsetRotation,Weapon->AttachmentOffsetLocation,ConvertedRotation, ConvertedLocation);

		FVector BoneSpaceLocation;

		FRotator BoneSpaceRotation;

		CharacterRef->GetMesh()->USkinnedMeshComponent::TransformFromBoneSpace(FName("GrabBone_r"), ConvertedLocation, ConvertedRotation, BoneSpaceLocation, BoneSpaceRotation);

		FTransform Transform(BoneSpaceRotation, BoneSpaceLocation, FVector(1, 1, 1));

		Weapon->SetActorTransform(Transform);

		CurrentWeaponSlot = Slot;

		Weapon->SetOwner(GetOwner());

		Weapon->SetEquipmentRef(this);

		CurrentWeapon = Weapon;

		PlayAnimMonCharacter(Weapon->PulloutAnim_Char);


		BindRecoilCurve(Weapon);


		WeaponAimOffset.Y = 0;

		WeaponAimOffset.X = 0;

		CharacterRef->GetAnimInstance()->UpdateWeaponInfo();

	}

}
```

```
void Ucpp_EquipmentComponent::SelectSlot(EWeaponSlotsEnum Slot)
{
	if (WeaponSlotsMap.Find(Slot))
	{
		if (CurrentWeaponSlot != Slot && bIsWeaponDrawn)
		{
			HolsterWeapon(true);
			PulloutWeapon(Slot);
		}
		else if (CurrentWeaponSlot != Slot)
		{
			PulloutWeapon(Slot);
		}
		else
		{
			HolsterWeapon(false);
		}
	}
}

void Ucpp_EquipmentComponent::NextFiremode()
{
	if (CurrentWeapon) {
		CurrentWeapon->CurrentFiremode++;
		if (CurrentWeapon->CurrentFiremode > CurrentWeapon->Firemodes.Num() - 1)
CurrentWeapon->CurrentFiremode = 0;
	}
}
void Ucpp_EquipmentComponent::FireButtonPressed()
```

```
{
	if (bSaveRotationBefore) {
		bSaveRotationBefore = false;
		RotationBeforeRecoil = CharacterRef->GetControlRotation();
	}
	Fire();
}

void Ucpp_EquipmentComponent::FireButtonReleased()
{
	if (CurrentWeapon) {
		CurrentWeapon->TriggerReleased();
	}
}

void Ucpp_EquipmentComponent::WeaponFiredCallback()
{
	bResetRecoil = false;
	bIsShooting = true;
	HipfireOffsetModified.X = UKismetMathLibrary::RandomFloatInRange(-CurrentWeapon->HipAccuracy, CurrentWeapon->HipAccuracy)*(1-AimDownsightsWeight);
	HipfireOffsetModified.Y = UKismetMathLibrary::RandomFloatInRange(-CurrentWeapon->HipAccuracy, CurrentWeapon->HipAccuracy) * (1 - AimDownsightsWeight);
	RecoilTimeline->SetPlayRate(1.0f);
	RecoilTimeline->Play();
}

void Ucpp_EquipmentComponent::WeaponStopFireCallback()
{
	RecoilTimeline->Stop();
```

```
	bIsShooting = false;
	if (bSaveRotationAfter) {
		bSaveRotationAfter = false;
		RotationAfterRecoil = CharacterRef->GetControlRotation();
	}
	ResetRecoil();
	RecoilTimeline->SetPlayRate(RecoilReverseTime);
	RecoilTimeline->Reverse();
}

void Ucpp_EquipmentComponent::ResetRecoil()
{
	bSaveRotationAfter = true;
	bSaveRotationBefore = true;
	bResetRecoil = true;
	bIsShooting = false;
}

void Ucpp_EquipmentComponent::SetDeltaMouseMovement(const FVector2D& inMouseDelta)
{
	if (bIsShooting) {
		if (inMouseDelta.Y > 0)
		{
			RecoilToCompensate = RecoilToCompensate - inMouseDelta.Y;
			if (RecoilToCompensate <= 0) RecoilToCompensate = 0;
			if (!bIsAimingDownsights)
			{
				OffsetRecoilToCompensate = OffsetRecoilToCompensate -
inMouseDelta.Y;
```

```
                        if (OffsetRecoilToCompensate <= 0) OffsetRecoilToCompensate = 0;
                    }
                }
                if (RecoilToCompensateH < 0 && inMouseDelta.X < 0)
                {
                    RecoilToCompensateH = RecoilToCompensateH - inMouseDelta.X;
                    if (!bIsAimingDownsights)
                    {
                        OffsetRecoilToCompensateH = OffsetRecoilToCompensateH +
inMouseDelta.X;
                    }
                }
                else
                if (RecoilToCompensateH > 0 && inMouseDelta.X > 0)
                {
                    RecoilToCompensateH = RecoilToCompensateH - inMouseDelta.X;
                    if (!bIsAimingDownsights)
                    {
                        OffsetRecoilToCompensateH = (OffsetRecoilToCompensateH +
inMouseDelta.X);
                    }
                }
                if ((RecoilToCompensateH >= -0.01) && (RecoilToCompensateH <= 0.01))
RecoilToCompensateH = 0;
        }
        OffsetWeapon(inMouseDelta);
}

void Ucpp_EquipmentComponent::Fire()
{
```

```
	if (CurrentWeapon)	CurrentWeapon->TriggerPressed();
}

void Ucpp_EquipmentComponent::Reload()
{
	if (CurrentWeapon)
	{
		if (CurrentWeapon->Reload()) {
			PlayAnimMonCharacter(CurrentWeapon->Reload_Tac_Char);
			CurrentWeapon->PlayAnimMonWeapon(CurrentWeapon->Reload_Tac_Gun);
		}
	}
}

void Ucpp_EquipmentComponent::AimDownsights()
{
	if (CurrentWeapon)
	{
		bIsEnteringDownSights = true;
		bIsAimingDownsights = true;
		bShouldChangeFOV = true;
		CharacterRef->GetAnimInstance()->UpdateWeaponInfo();
	}
}

void Ucpp_EquipmentComponent::ReleaseAimDownsights()
{
		bIsAimingDownsights = false;
```

```
		bShouldChangeFOV = true;
}

void Ucpp_EquipmentComponent::ScrollWheelUsed(const FInputActionValue& Value)
{
	if (CurrentWeapon)
	{
		float ScrollValue = Value.Get<float>();
		if (ScrollValue > 0)
		{
			ScrollWheelUp();
			return;
		}
		if (ScrollValue < 0)
		{
			ScrollWheelDown();
			return;
		}
	}
}

void Ucpp_EquipmentComponent::ScrollWheelUp()
{
	CurrentWeapon->CurrentSightInteger++;
	int MaxSightInteger = CurrentWeapon->GetSightsArraySize();
	if (CurrentWeapon->CurrentSightInteger >= MaxSightInteger) CurrentWeapon-
>CurrentSightInteger = MaxSightInteger-1;
	CharacterRef->GetAnimInstance()->CycleOptic();
	bIsChangingSights = true;
```

```
	bShouldChangeFOV = true;
}

void Ucpp_EquipmentComponent::ScrollWheelDown()
{
	CurrentWeapon->CurrentSightInteger--;
	if (CurrentWeapon->CurrentSightInteger < 0) CurrentWeapon->CurrentSightInteger = 0;
	CharacterRef->GetAnimInstance()->CycleOptic();
	bIsChangingSights = true;
	bShouldChangeFOV = true;
}
// RockInAHardPlace

#pragma once

#include "CoreMinimal.h"
#include "Components/ActorComponent.h"
#include "RockInAHardPlace/Enums/cpp_SlotEnums.h"
#include "Components/TimelineComponent.h"
#include "InputActionValue.h"
#include "cpp_EquipmentComponent.generated.h"

class Acpp_ItemWeapon;
class UTimelineComponent;
class Acpp_Mag;

UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class ROCKINAHARDPLACE_API Ucpp_EquipmentComponent : public UActorComponent
{
```

```
	GENERATED_BODY()


public:
	// Sets default values for this component's properties
	Ucpp_EquipmentComponent();
	friend class AcppRIHPCharacter;
	class AcppRIHPCharacter* CharacterRef;
	void Initialization(AcppRIHPCharacter* character);
	FVector2D WeaponAimOffset{};
	UPROPERTY(BlueprintReadOnly)
	float HipfireAlpha;


	//WeaponReload
	Acpp_Mag* NewMag;
	void GrabMag(int inWhichMag);
	void DropMag(int inWhichMag);
	void SpawnMag();
	void InsertMag();
	void DeleteMag();
	void ReloadFinish();


protected:
	// Called when the game starts
	virtual void BeginPlay() override;
private:
	bool bCanFire=true;
	UPROPERTY(EditDefaultsOnly, BluepRintReadWrite, Category = Character, meta =
(AllowPrivateAccess = "true"))
	TSubclassOf <class Acpp_ItemWeapon> PrimaryWeaponClass;
```

```
	Acpp_ItemWeapon* PrimaryWeaponSlot;

	UPROPERTY(EditDefaultsOnly, BluepRintReadWrite, Category = Character, meta =
(AllowPrivateAccess = "true"))

	TSubclassOf <class Acpp_ItemWeapon> SecondaryWeaponClass;

	Acpp_ItemWeapon* SecondaryWeaponSlot;

	UPROPERTY(EditDefaultsOnly, BluepRintReadWrite, Category = Character, meta =
(AllowPrivateAccess = "true"))

	TSubclassOf <class Acpp_ItemWeapon> SidearmWeaponClass;

	Acpp_ItemWeapon* SidearmWeaponSlot;


	TMap<EWeaponSlotsEnum, Acpp_ItemWeapon*> WeaponSlotsMap;

	EWeaponSlotsEnum CurrentWeaponSlot=EWeaponSlotsEnum::NullSlot;

	TMap<EWeaponSlotsEnum, FName> WeaponSocketsMap;

	Acpp_ItemWeapon* CurrentWeapon;


	bool bIsWeaponDrawn;


	void Fire();

	void Reload();

	void AimDownsights();

	void ReleaseAimDownsights();


	void ScrollWheelUsed(const FInputActionValue& Value);

	void ScrollWheelUp();

	void ScrollWheelDown();


	void OffsetWeapon(const FVector2D& inLook);


	//Socket names

	FName PrimarySocketName = FName("RifleInFrontSocket");
```

```
FName SecondarySocketName = FName("RifleOnBackSocket");

FName SidearmSocketName = FName("SidearmSocket");

FName HandSocket = FName("GrabBone_rSocket");


//Converts values recieved from Blender to unreal engine

//if objects flipped 180d back, means grabbone was flipped and code needs to be rewritten

void ConvertTransformValuesToUE(const FRotator& inRotator, const FVector&
inLocation, FRotator& outRotator, FVector& outLocation);


float CurrentZoom = 1.15f;

float ChangeSightTime = 0.05;

float AimDownsightsWeight = 0.0f;

bool bIsAimingDownsights;

bool bIsEnteringDownSights=false;

bool bShouldChangeFOV=false;

bool bIsChangingSights=false;


//Recoil

UTimelineComponent* RecoilTimeline;

FOnTimelineFloat RecoilVerticalInterpFun{};

FOnTimelineFloat RecoilHorizontalInterpFun{};

UTimelineComponent* RecoilReduceTimeline;

UFUNCTION()

void RecoilProgressVertical(float inRecoil);

UFUNCTION()

void RecoilProgressHorizontal(float inRecoil);

void BindRecoilCurve(Acpp_ItemWeapon* inWeapon);

FRotator RotationBeforeRecoil;

FRotator RotationAfterRecoil;
```

```
        bool bSaveRotationBefore=true;
        bool bSaveRotationAfter = true;
        bool bResetRecoil;
        UPROPERTY(BlueprintReadWrite, meta = (allowprivateaccess = "true"))
        bool bIsShooting;
        float RecoilToCompensate;
        float RecoilToCompensateH;
        float OffsetRecoilToCompensate;
        float OffsetRecoilToCompensateH;
        float RecoilReverseTime = 1.5f;
        FVector2D HipfireOffset{};
        FVector2D HipfireOffsetModified{};
        void CompensateOffsetRecoil();

public:
        // Called every frame
        virtual void TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction) override;

        void ChangeFOV(float DeltaTime);

        void PlayAnimMonCharacter(UAnimMontage* inAnimToPlay);
        void HolsterWeapon(bool bSwitchWeapon);
        void PulloutWeapon(EWeaponSlotsEnum Slot);
        void SelectSlot(EWeaponSlotsEnum Slot);
        void NextFiremode();

        void FireButtonPressed();
        void FireButtonReleased();
```

```
        void WeaponFiredCallback();


        void WeaponStopFireCallback();
        void ResetRecoil();
        void SetDeltaMouseMovement(const FVector2D& inMouseDelta);


        //Getters
        UFUNCTION(BlueprintCallable, BlueprintPure)
        bool GetIsWeaponDrawn() { return bIsWeaponDrawn; }
        UFUNCTION(BlueprintCallable, BlueprintPure)
        Acpp_ItemWeapon* GetCurrentWeapon() { return CurrentWeapon; }
        bool GetbIsAimingdownsights() { return bIsAimingDownsights; }
        float GetAimDownsightsWeight() { return AimDownsightsWeight; }
        const FVector2D& GetWeaponAimOffset() { return WeaponAimOffset; }
        UFUNCTION(BlueprintCallable, BlueprintPure)
        const FVector2D& GetHipfireOffset() { return HipfireOffset; }
};
```

Додаток Г – Графічна частина

ГРАФІЧНА ЧАСТИНА

РОЗРОБКА TRUE FIRST PERSON SHOOTER ГРИ "ROCK IN A HARD PLACE: LONESOME ROAD" З ВИКОРИСТАННЯМ ПЛАТФОРМИ UNREAL ENGINE 5

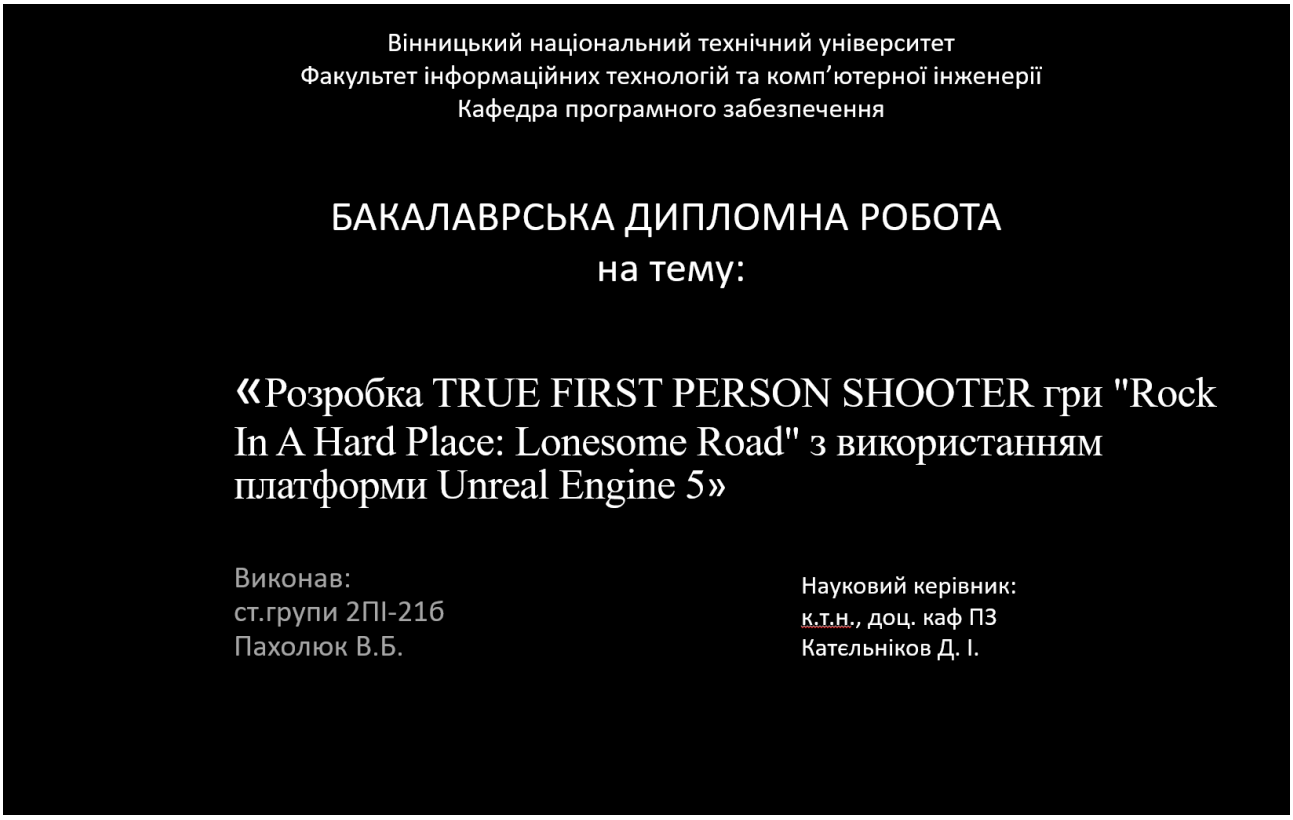

Рисунок Г.1 – Титульний слайд

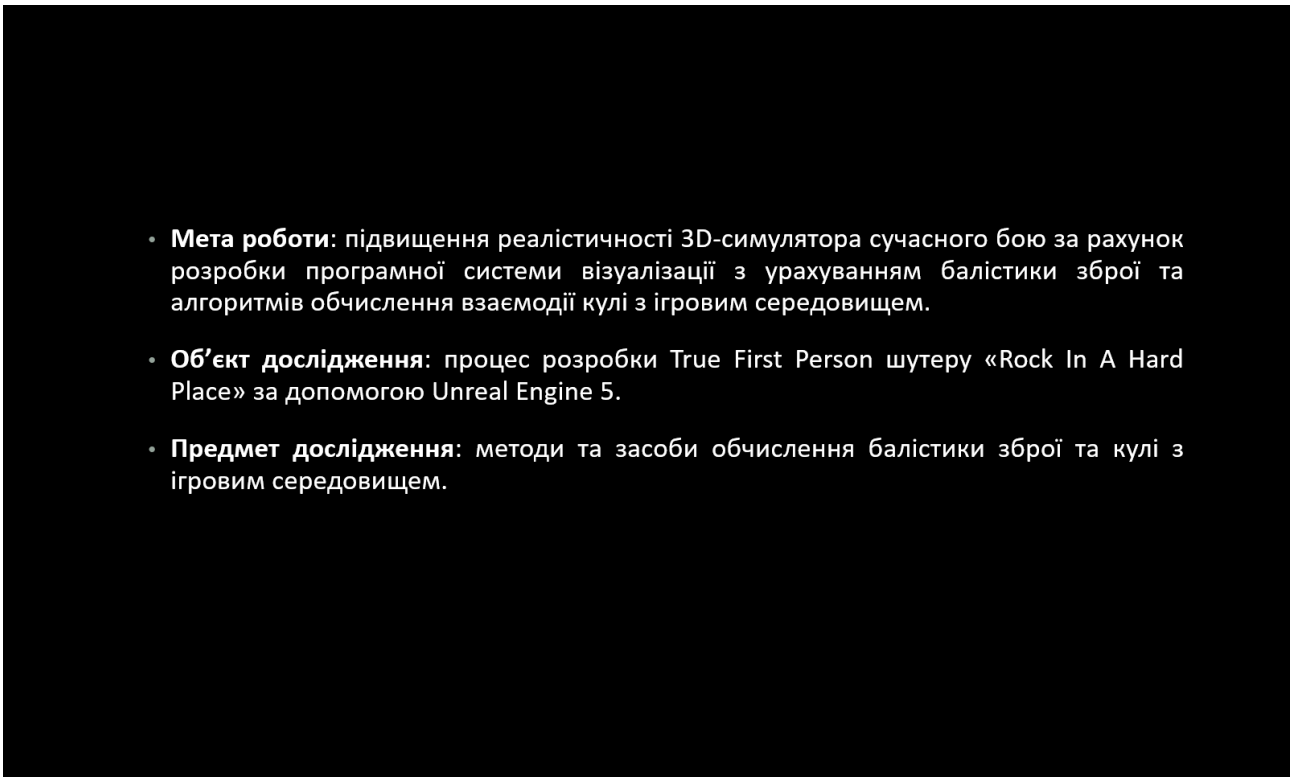

Рисунок Г.2 – Мета роботи

- **Наукова новизна отриманих результатів.** Подальшого розвитку набув метод розробки True First Person-шутерів, який, на відміну від існуючих, не лише імітує постріл візуальним ефектом, а й виконує високоточні розрахунки траєкторій куль, моделює віддачу зброї, знос та інші аспекти, що забезпечують глибше занурення гравця у віртуальне середовище.
- **Практична цінність отриманих результатів.** Практична цінність одержаних результатів полягає в тому, що на основі отриманих в бакалаврській кваліфікаційної роботі теоретичних положень запропоновано алгоритми та розроблено програмні засоби відеогри «Rock In A Hard Place».

Рисунок Г.3 – Наукова новизна

Порівняльний аналіз аналогів

Критерії	Call of Duty	Battlefield	Counter-Strike: Global Offensive	DOOM	Rainbow Six Siege	Rock In A Hard Place
Можливість змінювати оптику	+	-	-	-	-	+
Симуляція балістики	-	+	-	-	-	+
Вид від справжньої першої особи	-	-	-	-	-	+
Динамічне управління персонажем	-	-	-	-	+	+
Можливість прицілювання	-	-	-	-	+	+
Загальна оцінка	20%	20%	0%	0%	40%	100%

Рисунок Г.4 – Порівняльний аналіз аналогів

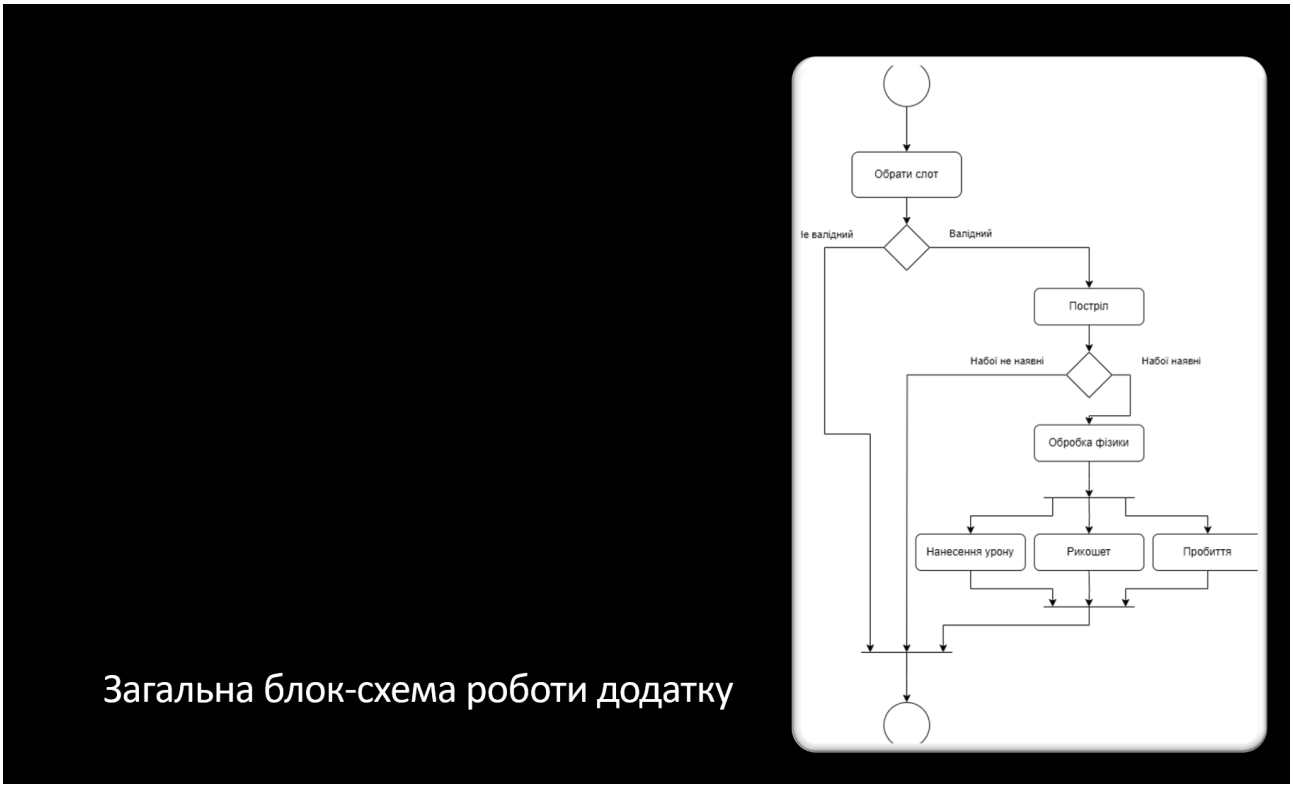

Рисунок Г.5 – Блок-схема

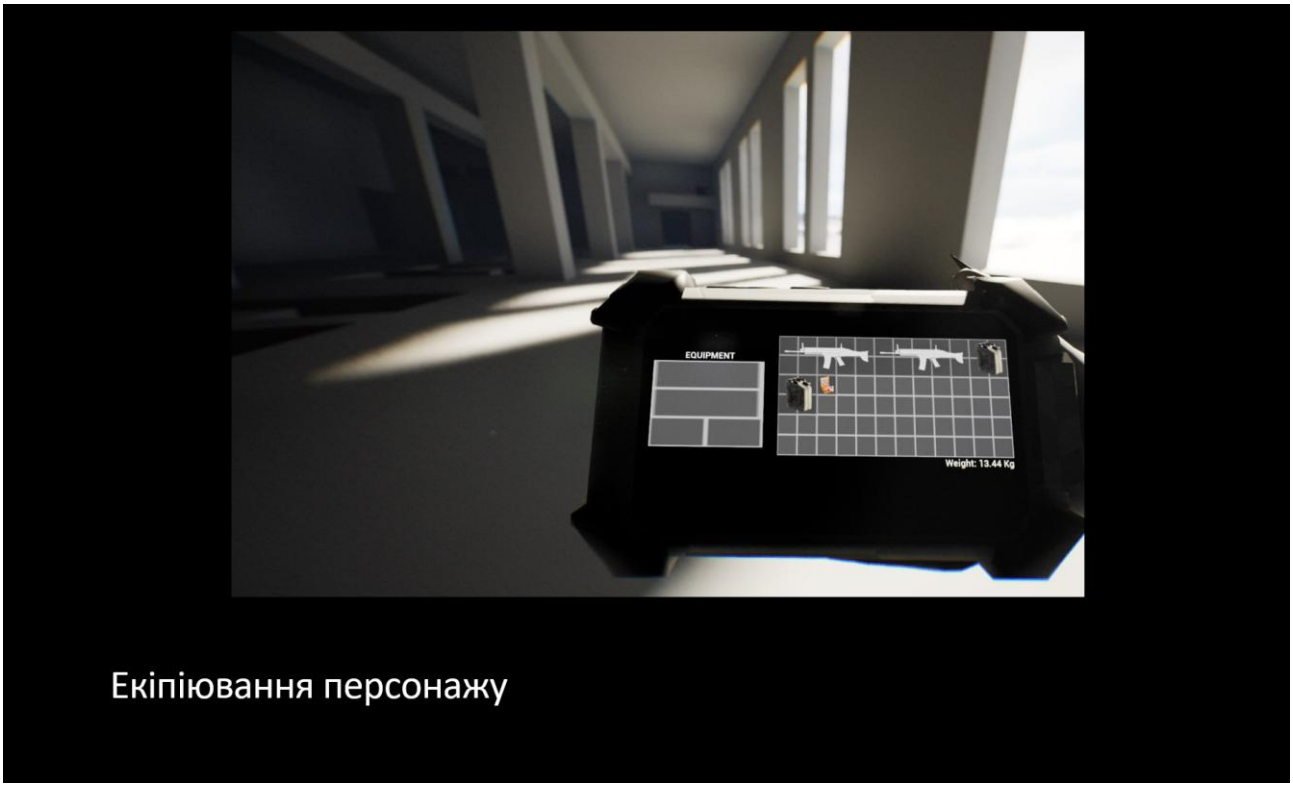

Рисунок Г.6 – Екіпіювання персонажу

Рисунок Г.7 – Стрільба

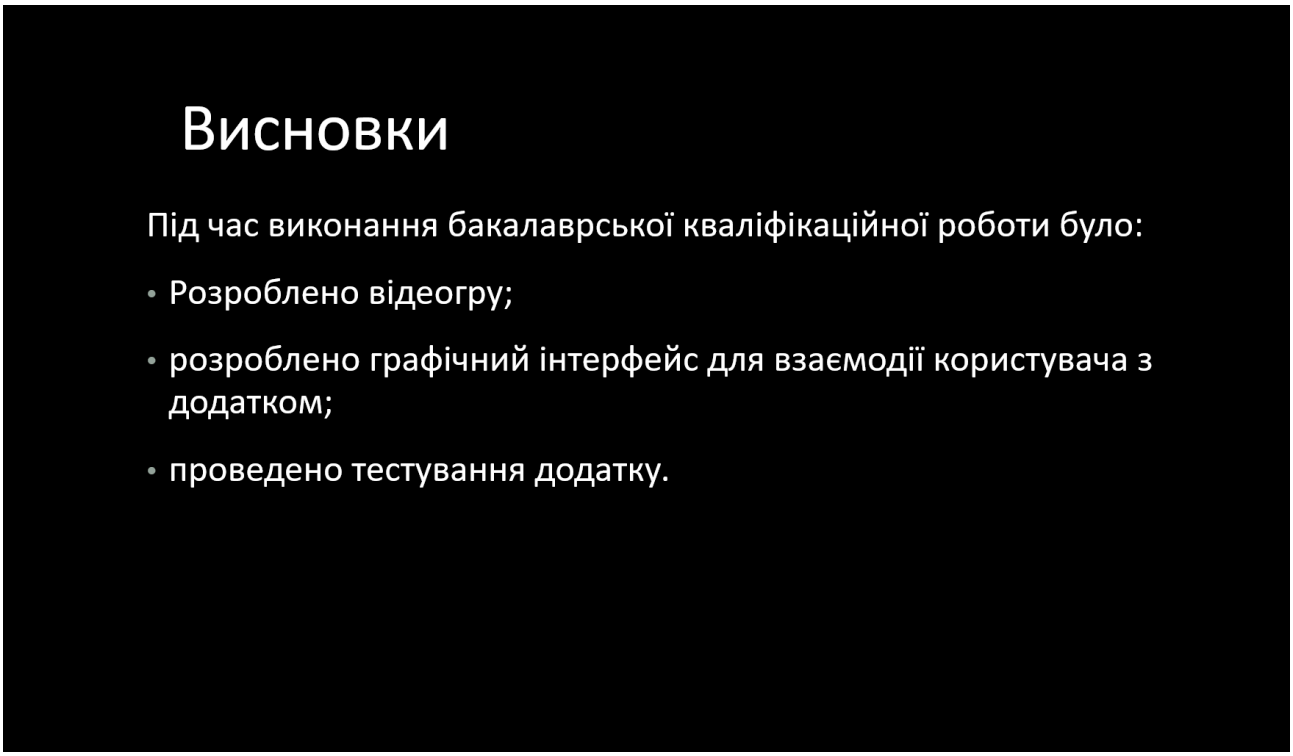

Рисунок Г.8 – Висновки

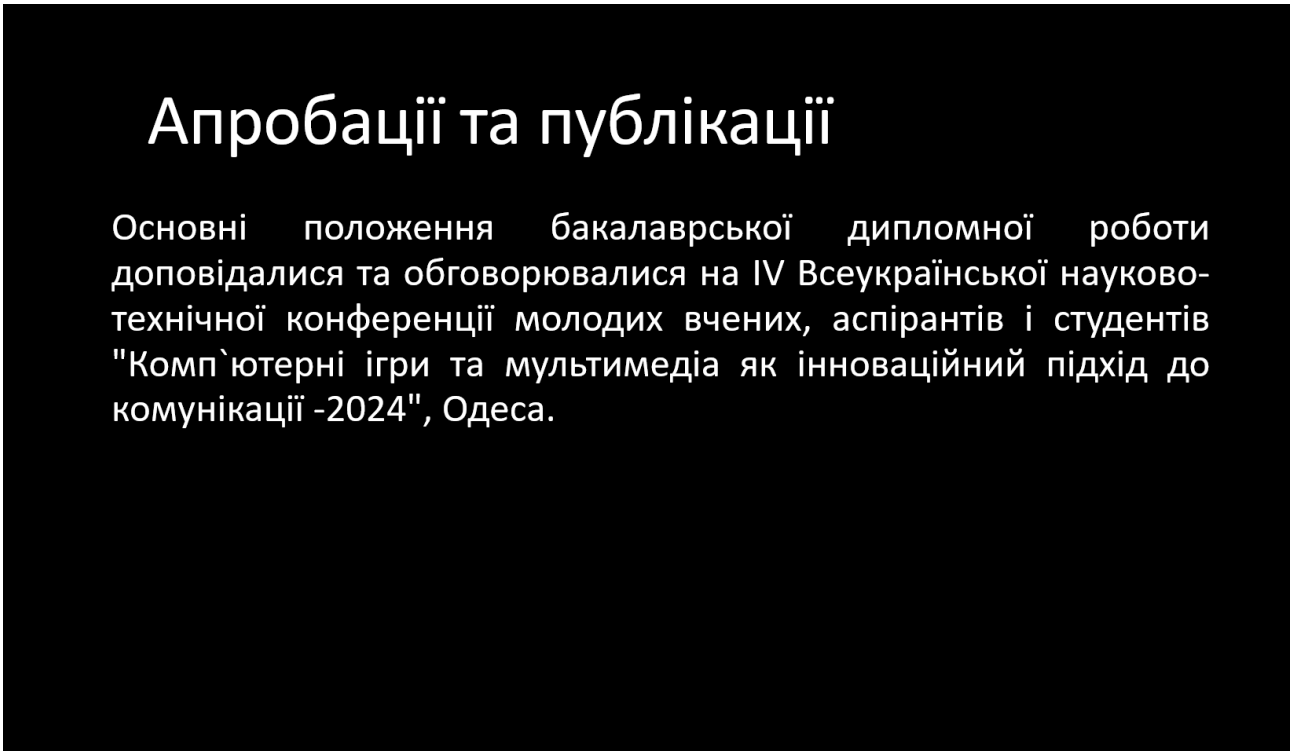

Рисунок Г.9 – Публікації